デベロッパーガイド、バージョン 2

AWS IoT Greengrass

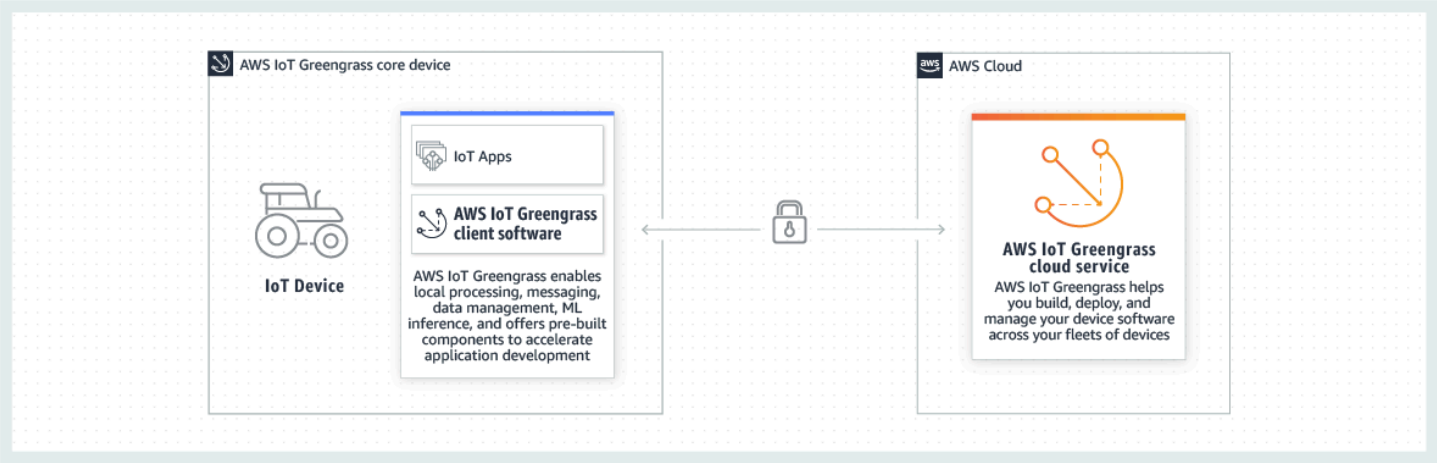
AWS IoT Greengrass: デベロッパーガイド、バージョン 2

Table of Contents

AWS IoT Greengrass とは

AWS IoT Greengrass は、デバイスで IoT アプリケーションを構築、デプロイ、管理するうえで役立つオープンソースのモノのインターネット (IoT) エッジランタイムとクラウドサービスです。AWS IoT Greengrass を使用して、デバイスが生成するデータに対するローカルに動作、機械学習モデルに基づいて予測の実行、デバイスデータのフィルタリングと集計を実行できるソフトウェアを構築できます。AWS IoT Greengrass は、デバイスがデータの生成場所に近いデータの収集と分析、ローカルイベントに自律的に反応、ローカルネットワークの他のデバイスと安全に通信できるようにします。Greengrass デバイスも AWS IoT Core と安全に通信して、IoT データを AWS クラウド にエクスポートできます。AWS IoT Greengrass を使用し、AWS サービスまたはサードパーティーのサービスにエッジデバイスを接続するプレビルド ソフトウェア モジュール (コンポーネントと呼ばれる) を使用して、エッジアプリケーションを構築します。Lambda 関数、Docker コンテナ、ネイティブ オペレーティング システムのプロセス、任意のカスタムランタイムを使用して、ソフトウェアをパッケージ化して実行する AWS IoT Greengrass も使用できます。

次の例では、AWS IoT Greengrass デバイスが AWS クラウド と対話する方法を示します。



新機能

AWS IoT Greengrass V2 では、新機能と改善点が導入されています。バージョン 2 で提供される新機能の詳細を以下に示します。

- の最新情報 AWS IoT Greengrass Version 2

AWS IoT Greengrass を初めてお使いになるユーザーの場合

AWS IoT Greengrass を初めて使用する場合、次のセクションをお読みになることをお勧めします:

- AWS IoT Greengrass の働き

次に、入門チュートリアルに従って、AWS IoT Greengrass の基本的な機能を試してみます。このチュートリアルでは、デバイスに AWS IoT Greengrass Core ソフトウェアのインストール、Hello World コンポーネントの開発、デプロイ用にそのコンポーネントをパッケージ化します。

AWS IoT Greengrass の既存ユーザーの場合

の現在のユーザーにはAWS IoT Greengrass V1、Greengrass バージョン 1 と Greengrass バージョン 2 の違いを理解し、バージョン 1 からバージョン 2 に移行する方法を理解するために、以下のトピックをお勧めします。

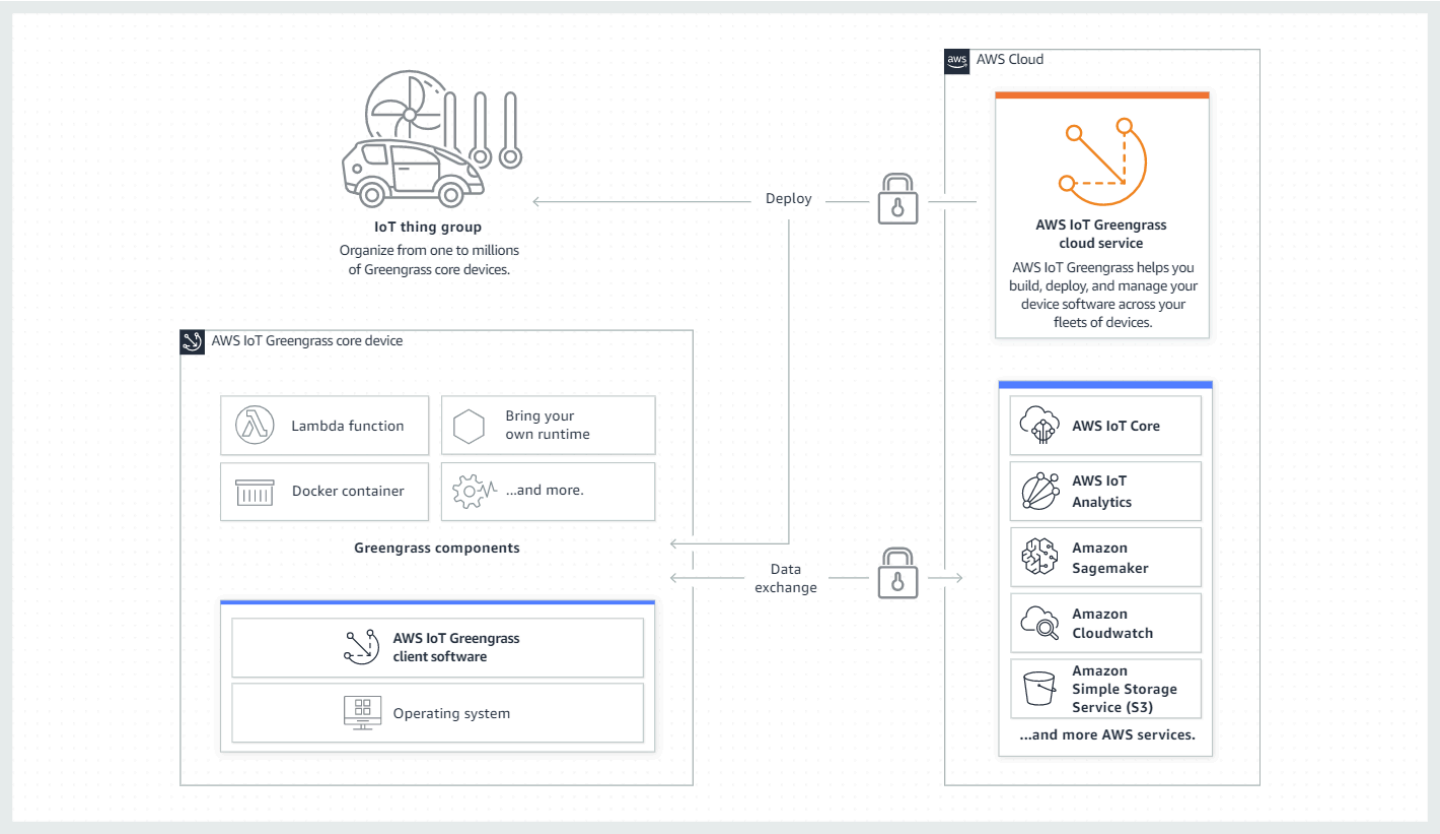
- AWS IoT Greengrass バージョン 1 から移行

AWS IoT Greengrass の働き

AWS IoT Greengrass クライアントソフトウェアはAWS IoT Greengrass Core ソフトウェアとも呼ばれ、ARM または x86 アーキテクチャを持つデバイス向けの、Windows および Linux ベースのディストリビューション (Ubuntu や Raspberry Pi OS など) で動作します。AWS IoT Greengrass を使用すると、デバイスが生成するデータに対してローカルに動作するようプログラミングを行い、機械学習モデルに基づく予測を実行し、デバイスデータのフィルタリングおよび集計を行うことができます。AWS IoT Greengrass は AWS Lambda 関数、Docker コンテナ、ネイティブ OS プロセス、または任意のカスタムランタイムのローカル実行を可能にします。

AWS IoT Greengrass には、エッジデバイスの機能を簡単に拡張できる、事前に構築されたソフトウェアモジュール (コンポーネントと呼ばれます) が用意されています。AWS IoT Greengrass コンポーネントを使用すると、エッジでAWS サービスおよびサードパーティ製アプリケーションに接続できます。IoT アプリケーションの開発後、AWS IoT Greengrass は現場のデバイスフリートにおいてこれらのアプリケーションをリモートでデプロイ、設定、および管理することを可能にします。

以下の例は、AWS IoT Greengrass デバイスが AWS IoT Greengrass のクラウドサービスおよび AWS クラウド のその他の AWS サービスとどのように連携するかを示しています。



AWS IoT Greengrass の主要なコンセプト

AWS IoT Greengrass を理解して使用する上で不可欠な概念を以下に示します:

AWS IoT モノ

AWS IoT モノは、特定のデバイスまたは論理エンティティを表します。モノに関する情報は AWS IoTレジストリに保存されます。

Greengrass コアデバイス

AWS IoT Greengrass Core ソフトウェアを実行するデバイス。Greengrass コアデバイスは AWS IoT モノです。複数のCore デバイスを AWS IoT モノグループに追加して、Greengrass コアデバイスのグループを作成および管理できます。詳細については、「AWS IoT Greengrass コアデバイスをセットアップする」を参照してください。

Greengrass クライアントデバイス

MQTT を介して Greengrass コアデバイスに接続して通信するデバイス。Greengrass クライアントデバイスは AWS IoT モノです。コアデバイスは、接続されたクライアントデバイスからのデータの処理、フィルタリング、および集計を行うことができます。クライアントデバイス、AWS IoT Core クラウドサービス、Greengrass コンポーネントの間で MQTT メッセージを

リレーするように、コアデバイスを設定できます。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

クライアントデバイスは FreeRTOS を実行、または AWS IoT Device SDK またはGreengrass 検出 API を使用して、接続可能なコアデバイスに関する情報を取得できます。

Greengrass コンポーネント

Greengrass コアデバイスにデプロイされ実行されるソフトウェアモジュールです。AWS IoT Greengrass で開発およびデプロイされるソフトウェアはすべてコンポーネントとしてモデル化されます。AWS IoT Greengrass は構築済みのパブリックコンポーネントを提供し、これによりアプリケーションで使用できる機能が提供されます。ローカルデバイスまたはクラウドで、独自のカスタムコンポーネントを開発することもできます。カスタムコンポーネント開発後は、AWS IoT Greengrass クラウドサービスを使用して、単一または複数のコアデバイスにコンポーネントをデプロイできます。カスタムコンポーネントを作成して、そのコンポーネントをコアデバイスにデプロイできます。これを行うとき、コアデバイスはコンポーネントを実行するために、以下のリソースをダウンロードします。

- レシピ: コンポーネントの詳細、設定、およびパラメータを定義してソフトウェアモジュールを記述する JSON または YAML ファイル。
- アーティファクト: デバイスで実行するソフトウェアを定義するソースコード、バイナリ、またはスクリプト。アーティファクトをゼロから作成することも、Lambda 関数、Docker コンテナ、またはカスタムランタイムを使用してコンポーネントを作成することもできます。
- 依存関係: コンポーネント間の関係で、依存コンポーネントの自動更新または再起動を強制できるもの。たとえば、暗号化コンポーネントに依存させることにより、セキュアなメッセージ処理コンポーネントを得ることができます。これにより、暗号化コンポーネントが更新されると、メッセージ処理コンポーネントも自動的に更新され再起動されるようになります。

詳細については、「AWS が提供したコンポーネント と AWS IoT Greengrass コンポーネントを開発する」を参照してください。

デプロイメント

コンポーネントを送信し、目的のコンポーネント設定を宛先ターゲットデバイスに適用するプロセス。ターゲットデバイスには、単一の Greengrass コアデバイスまたは Greengrass コアデバイスのグループを指定できます。デプロイでは、更新されたコンポーネント設定がターゲットに自動的に適用されます。依存関係が定義されているその他のコンポーネントもこの処理に含まれます。既存のデプロイをクローンして、同じコンポーネントを使用しつつ別のターゲットにデプロイされる新しいデプロイを作成することもできます。デプロイは連続的なものです。つまり、デプロイによるコンポーネントまたはコンポーネント設定に対する更新は、自動的にすべての

ターゲットに送信されます。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

AWS IoT Greengrass Core ソフトウェア

コアデバイスにインストールするすべての AWS IoT Greengrass ソフトウェアのセットです。AWS IoT GreengrassCore ソフトウェアの設定は次のとおりです。

- nucleus: このコンポーネントは必須であり、AWS IoT Greengrass Core ソフトウェアの最小限の機能を提供します。nucleus は他のコンポーネントのデプロイ、オーケストレーション、およびライフサイクル管理を管理します。また、個々のデバイスで AWS IoT Greengrass コンポーネント間のローカルなコミュニケーションを円滑化します。詳細については、「Greengrass nucleus」を参照してください。
- オプションのコンポーネント: これらの設定可能なコンポーネントは AWS IoT Greengrass によって提供され、エッジデバイスの追加機能を有効にします。必要に応じて、データストリーミング、ローカル機械学習推論、ローカルコマンドラインインターフェイスなど、デバイスにデプロイするオプションコンポーネントを選択できます。詳細については、「AWS が提供したコンポーネント」を参照してください。

コンポーネントの新しいバージョンをデバイスにデプロイすることにより、AWS IoT Greengrass Core ソフトウェアをアップグレードできます。

AWS IoT Greengrass の機能

AWS IoT Greengrass Version 2 は以下の要素で設定されています。

- ソフトウェアディストリビューション
  - AWS IoT Greengrass Core ソフトウェアのインストールに最低限必要となる、Greengrass nucleus コンポーネントです。このコンポーネントは、Greengrass コンポーネントのデプロイ、オーケストレーション、ライフサイクル管理を管理します。
  - サービス、プロトコル、およびソフトウェアと統合するために、追加オプションとして AWS が提供するコンポーネントです。
  - カスタム Greengrass コンポーネントの作成、テスト、構築、パブリッシュ、デプロイに使用する、Greengrass 開発ツールです。
  - クライアントデバイスで使用する、カスタム Greengrass コンポーネントと Greengrass Discovery ライブラリ用のプロセス間通信 (IPC) ライブラリを含む、AWS IoT Device SDK です。
  - コアデバイスでデータストリームの管理に使用する、ストリームマネージャー SDK です。

- クラウドサービス
  - AWS IoT Greengrass V2 API
  - AWS IoT Greengrass V2 コンソール

AWS IoT Greengrass Core ソフトウェア

エッジデバイスで実行可能な AWS IoT Greengrass Core ソフトウェアを使用して、以下を実行できます。

- AWS クラウドへの自動エクスポートにより、データストリームをローカルデバイスで処理します。詳細については、「Greengrass コアデバイスでのデータストリームの管理」を参照してください。
- AWS IoT とコンポーネント間の MQTT メッセージングをサポートします。詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。
- MQTT 経由で接続および通信を行うローカルデバイスとやり取りします。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。
- コンポーネント間のローカルパブリッシュおよびサブスクライブメッセージをサポートします。詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。
- コンポーネントと Lambda 関数をデプロイして呼び出します。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。
- インストールスクリプトや実行スクリプトのサポートなどにより、コンポーネントのライフサイクルを管理します。詳細については、「AWS IoT Greengrass コンポーネントレシピのリファレンス」を参照してください。
- AWS IoT Greengrass Core ソフトウェアとカスタムコンポーネントの安全な over-the-air (OTA) ソフトウェア更新を実行します。詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新 と デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。
- ローカルシークレットの安全な暗号化されたストレージおよびコンポーネントによる制御されたアクセスを提供します。詳細については、「シークレットマネージャー」を参照してください。
- デバイスの認証と承認による、デバイスと AWS クラウド間の安全な接続。詳細については、「AWS IoT Greengrass のデバイス認証と認可」を参照してください。

Greengrass コアデバイスの設定と管理は AWS IoT Greengrass API で行います。ここでは、継続的なソフトウェアのデプロイの作成を行います。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

一部の機能は、特定のプラットフォームのみでサポートされています。詳細については、「オペレーティングシステム別 Greengrass 機能の互換性」を参照してください。

サポートされるプラットフォーム、要件、およびダウンロードの詳細については、「AWS IoT Greengrass コアデバイスをセットアップする」を参照してください。

このソフトウェアをダウンロードすると、Greengrass Core ソフトウェアのライセンス契約に同意したものと見なされます。

オペレーティングシステム別 Greengrass 機能の互換性

AWS IoT Greengrass は、さまざまなオペレーティングシステムを実行するデバイスをサポートします。一部の機能は、特定のオペレーティングシステムでのみサポートされています。次の表を使用して、サポートされている各オペレーティングシステムで利用できる機能を確認してください。サポートされているオペレーティングシステム、要件、Greengrass コアデバイスの設定方法の詳細については、「AWS IoT Greengrass コアデバイスをセットアップする」を参照してください。

メッセージング

機能	Linux	Windows
AWS IoT と コンポーネント間での MQTT メッセージの交換	 はい	 はい
コンポーネント間でローカルのパブリッシュ/サブスクライブメッセージ交換	 はい	 はい
MQTT 経由でローカル IoT デバイスとやり取りする	 はい	 はい

機能	Linux	Windows
Modbus-RTU コンポーネントを使用して、ローカル Modbus-RTU デバイスとやり取り	 はい	 いいえ

セキュリティ

機能	Linux	Windows
デバイス認証と認可で接続を保護	 はい	 はい
から安全な暗号化されたシークレットをデプロイしてアクセスする AWS Secrets Manager	 はい	 はい
ハードウェアセキュリティモジュール (HSM) を使用して、デバイスのプライベートキーと証明書を安全に保管	 はい	 いいえ
を使用したコアデバイスの監査 AWS IoT Device Defender	 はい	 はい
AWS 認証情報を使用して AWS サービスとやり取りする	 はい	 はい

インストール

機能	Linux	Windows
自動プロビジョニング AWS IoT Greengrass を使用してインストールする	 はい	 はい
手動プロビジョニング AWS IoT Greengrass でインストールする	 はい	 はい
AWS IoT フリートプロビジョニング AWS IoT Greengrass を使用してインストールする	 はい	 はい
カスタムプロビジョニングプラグイン AWS IoT Greengrass を使用してインストールする	 はい	 はい
構築済みの AWS IoT Greengrass Docker イメージを使用して Docker コンテナで を実行する	 はい	 いいえ

リモートメンテナンスと更新

機能	Linux	Windows
安全な over-the-air (OTA) ソフトウェア更新を実行する	 はい	 はい

機能	Linux	Windows
でコアデバイスを管理する AWS Systems Manager	 はい	 いいえ
AWS IoT セキュアトンネリングを使用してコアデバイスに接続する	 はい	 いいえ

Machine Learning

機能	Linux	Windows
Amazon SageMaker Edge Manager を使用して機械学習推論を実行する	 はい	 はい
Amazon Lookout for Vision を使用して機械学習推論を実行する	 はい	 いいえ
DLR を使用して機械学習の推論を実行	 はい	 はい
を使用した機械学習推論の実行 TensorFlow	 はい	 はい

コンポーネント機能

機能	Linux	Windows
Lambda 関数のデプロイと呼び出し	 はい	 いいえ
コンポーネントで Docker コンテナを実行	 はい	 はい
ストリームマネージャーを使用して大量のデータストリームを処理とエクスポート	 はい	 はい
ライフサイクルスクリプトでコンポーネントのライフサイクルを管理する	 はい	 はい
デバイスシャドウとやり取り	 はい	 はい
Amazon CloudWatch Logs へのログのアップロード	 はい	 はい
CloudWatch メトリクスコンポーネントを使用して Amazon CloudWatch メトリク	 はい	 はい

機能	Linux	Windows
スにデータをアップロードする		
Amazon SNS コンポーネントを使用して Amazon Simple Notification Service にメッセージをパブリッシュ	 はい	 いいえ
ストリームマネージャーを使用して Amazon Data Firehose 配信ストリームにデータを発行する	 はい	 はい
Firehose コンポーネントを使用して Amazon Data Firehose 配信ストリームにデータを発行する	 はい	 いいえ
リアルタイムシステムのテレメトリメトリクスを収集して対応する	 はい	 はい
コンポーネントプロセスのシステムリソース制限を設定	 はい	 いいえ
コンポーネントプロセスを一時停止と再開	 はい	 いいえ

機能	Linux	Windows
AWS IoT SiteWise コンポーネント AWS IoT SiteWise を使用して と統合する	 はい	 はい
Kinesis Video Streams コンポーネントのエッジコネクタを使用して Amazon Kinesis Video Streams にビデオストリームをパブリッシュ	 はい	 いいえ

コンポーネントの開発

機能	Linux	Windows
コアデバイスにローカルでコンポーネントを開発	 はい	 はい
AWS IoT Greengrass CLI を使用してコアデバイスを操作する	 はい	 はい
ローカルデバッグコンソールを使用してコアデバイスとやり取り	 はい	 はい
カスタムコンポーネントで AWS IoT Device SDK for Python を使用する	 はい	 はい

機能	Linux	Windows
カスタムコンポーネントで AWS IoT Device SDK for C++ を使用する	 はい	 はい
カスタムコンポーネントで AWS IoT Device SDK for Java を使用する	 はい	 はい

デバイス証明書

機能	Linux	Windows
AWS IoT Device Tester AWS IoT Greengrass V2 を使用して IoT デバイスを検証する	 はい	 はい

の最新情報 AWS IoT Greengrass Version 2

AWS IoT Greengrass Version 2 は、以下の機能を導入 AWS IoT Greengrass した のメジャーバージョンです。

- パブリッシャーがサポートするコンポーネント – パブリッシャーがサポートするコンポーネントが提供される AWS IoT Greengrass ようになりました。これらのコンポーネントは、サードパーティーベンダーによって開発、提供、および提供されます。詳細については、「パブリッシャーがサポートするコンポーネント」を参照してください。
- VPC で Greengrass デバイスを運用する — VPC で Greengrass コアデバイスを運用できるようになりました。これにより、パブリックインターネットアクセスのない VPC でデプロイを実行できます。詳細については、「VPC で AWS IoT Greengrassコアデバイスを運用する」を参照してください。
- Greengrass Testing Framework (GTF) – の AWS IoT Greengrass Version 2 GTF が利用可能になりました。GTF は、 end-to-end オートメーションをサポートするための構成要素のコレクションです。これにより、 AWS IoT Greengrass Version 2 内部のお客様は、サービスチームがソフトウェアの変更、自動承認、品質保証の目的で使用しているのと同じテストフレームワークを使用できます。詳細については、Github の「Greengrass テストフレームワーク」を参照してください。
- PSA 認定 - AWS IoT Greengrass nucleus バージョン 2.7.0 以降は、Platform Security Architecture (PSA) 認定を受けています。詳細については、「AWS IoT Greengrass は PSA 認定を取得しました」を参照してください。

AWS IoT Greengrass リリースノートには、新機能、更新と改善、一般的な修正 AWS IoT Greengrass など、リリースに関する詳細が記載されています。 には、次のタイプのリリース AWS IoT Greengrass があります。

- の新機能リリース AWS IoT Greengrass
- AWS IoT Greengrass Core ソフトウェアの更新

このセクションには、すべての AWS IoT Greengrass V2 リリースノートと最新のリリースノートが含まれており、主要な機能変更と大幅なバグ修正が含まれています。その他のマイナー修正については、「」の「aws-greengrass organization」を参照してください GitHub。

リリースノート

- リリース: 2024 年 2 月 15 日 AWS IoT Greengrass Core v2.12.2 ソフトウェア更新

- リリース: 2023 年 12 月 8 日の AWS IoT Greengrass Core v2.12.1 ソフトウェア更新
- リリース: 2023 年 11 月 7 日の AWS IoT Greengrass Core v2.12.0 ソフトウェア更新
- リリース: 2023 年 10 月 18 日の AWS IoT Greengrass Core v2.11.3 ソフトウェア更新
- リリース: 2023 年 8 月 9 日 AWS IoT Greengrass Core v2.11.2 のソフトウェア更新
- リリース: 2023 年 7 月 21 日 AWS IoT Greengrass Core v2.11.1 ソフトウェア更新
- リリース: 2023 年 6 月 28 日、AWS IoT Greengrass Core v2.11.0 ソフトウェア更新
- リリース: 2023 年 6 月 21 日、AWS IoT Greengrass Core v2.10.3 ソフトウェア更新
- リリース: 2023 年 6 月 5 日、AWS IoT Greengrass Core v2.10.2 ソフトウェア更新
- リリース: 2023 年 5 月 11 日 AWS IoT Greengrass Core v2.10.1 ソフトウェア更新
- リリース: 2023 年 5 月 9 日 AWS IoT Greengrass Core v2.10.0 ソフトウェア更新
- リリース: 2023 年 4 月 20 日 AWS IoT Greengrass Core v2.9.6 ソフトウェア更新
- リリース: 2023 年 3 月 30 日 AWS IoT Greengrass Core v2.9.5 ソフトウェア更新
- リリース: 2023 年 2 月 24 日 AWS IoT Greengrass Core v2.9.4 ソフトウェア更新
- リリース: 2023 年 2 月 1 日 AWS IoT Greengrass Core v2.9.3 ソフトウェア更新
- リリース: 2022 年 12 月 22 日 AWS IoT Greengrass Core v2.9.2 ソフトウェア更新
- リリース: 2022 年 11 月 18 日 AWS IoT Greengrass Core v2.9.1 ソフトウェア更新
- リリース: 2022 年 11 月 15 日 AWS IoT Greengrass Core v2.9.0 ソフトウェア更新
- リリース: 2022 年 10 月 13 日、AWS IoT Greengrass Core v2.8.1 ソフトウェアの更新
- リリース: 2022 年 10 月 7 日、AWS IoT Greengrass Core v2.8.0 ソフトウェアの更新
- リリース: 2022 年 7 月 28 日 AWS IoT Greengrass Core v2.7.0 ソフトウェア更新
- リリース: 2022 年 6 月 27 日、AWS IoT Greengrass Core v2.6.0 ソフトウェアの更新
- リリース: 2022 年 5 月 31 日 AWS IoT Greengrass Core v2.5.6 ソフトウェア更新
- リリース: 2022 年 4 月 6 日 AWS IoT Greengrass Core v2.5.5 ソフトウェア更新
- リリース: 2022 年 3 月 23 日 AWS IoT Greengrass Core v2.5.4 ソフトウェア更新
- リリース: 2022 年 1 月 6 日 AWS IoT Greengrass Core v2.5.3 ソフトウェア更新
- リリース: 2021 年 12 月 3 日 AWS IoT Greengrass Core v2.5.2 ソフトウェア更新
- リリース: 2021 年 11 月 23 日 AWS IoT Greengrass Core v2.5.1 ソフトウェア更新
- リリース: 2021 年 11 月 12 日 AWS IoT Greengrass Core v2.5.0 ソフトウェア更新

- リリース: 2021 年 8 月 3 日 AWS IoT Greengrass Core v2.4.0 のソフトウェア更新
- リリース: 2021 年 6 月 29 日 AWS IoT Greengrass Core v2.3.0 ソフトウェア更新
- リリース: 2021 年 6 月 18 日 AWS IoT Greengrass Core v2.2.0 ソフトウェア更新
- リリース: 2021 年 4 月 26 日 AWS IoT Greengrass Core v2.1.0 ソフトウェア更新
- リリース: 2021 年 3 月 9 日 AWS IoT Greengrass Core v2.0.5 ソフトウェア更新
- リリース: 2021 年 2 月 4 日 AWS IoT Greengrass Core v2.0.4 ソフトウェア更新

リリース: 2024 年 2 月 15 日 AWS IoT Greengrass Core v2.12.2 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.12.2 と、 AWSが提供するコンポーネントの更新が提供されます。

リリース日： 2024 年 2 月 15 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次の表は、 が提供する、新機能と更新された機能 AWS を含むコンポーネントの一覧です。

> **Important**
>
> コンポーネントをデプロイすると、 はそのコンポーネントのすべての依存関係の最新のサポート対象バージョン AWS IoT Greengrass をインストールします。このため、モノのグループに新しいデバイスを追加したり、それらのデバイスを対象とするデプロイを更新したりすると、 AWSが提供するパブリックコンポーネントの新しいパッチバージョンがコアデバイスに自動的にデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。
> AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「」を参照してくださいAWS IoT Greengrass Core ソフトウェア (OTA) の更新。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.12.2 が利用可能になりました。 バグ修正と機能向上 • 古いログが正しくクリーンアップされない問題を修正しました。 • 一般的なバグ修正と機能強化。
シャドウマネージャー	シャドウマネージャーコンポーネントのバージョン 2.3.6 を利用できます。 バグ修正と機能向上 デバイスのオフライン中に AWS クラウド 更新によって削除されたシャドウプロパティが、接続の回復後もローカルシャドウに引き続き存在する問題を修正しました。
Lambda ランチャー	Lambda ランチャーコンポーネントのバージョン 2.0.13 を利用できます。 バグ修正と機能向上 一般的なバグ修正と機能強化。
ディスクスプーラ	ディスクスプーラコンポーネントのバージョン 1.0.3 を利用できます。 バグ修正と機能向上 データベース接続を再利用することでパフォーマンスが向上します。

リリース: 2023 年 12 月 8 日の AWS IoT Greengrass Core v2.12.1 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.12.1 と、 AWSが提供するコンポーネントの更新が提供されます。

リリース日： 2023 年 12 月 8 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.12.1 が利用可能になりました。 バグ修正と機能向上 • nucleus がデプロイトピックへの MQTT サブスクリプションを複製して、追加のログ記録や MQTT 発行が発生する可能性がある問題を修正しました。
クライアントデバイス認証	クライアントデバイス認証コンポーネントのバージョン 2.4.5 を利用できます。 新機能 `selectionRule` パラメータでモノの名前を選択するためのワイルドカードプレフィックスのサポートを追加しました。

コンポーネント	詳細
	バグ修正と機能向上 特定のケースで証明書が新しい接続情報で更新されない問題を修正しました。
ディスクスプーラ	ディスクスプーラコンポーネントのバージョン 1.0.2 を利用できます。 バグ修正と機能向上 特定のケースで MQTT メッセージ形式フィールドが保持されない問題を修正しました。
MQTT ブリッジ	ディスクスプーラコンポーネントのバージョン 2.3.1 を利用できます。 バグ修正と機能向上 ローカル MQTT クライアントが切断ループに入る問題を修正しました。
ストリームマネージャー	ストリームマネージャーコンポーネントのバージョン 2.1.12 を利用できます。 ??? バグ修正と機能向上 認証情報を使用する順序を更新して、AWSサービスリクエストに Greengrass 認証情報が優先されるようにします。

リリース: 2023 年 11 月 7 日の AWS IoT Greengrass Core v2.12.0 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.12.0 と、 AWSが提供するコンポーネントの更新が提供されます。

リリース日： 2023 年 11 月 7 日

リリースハイライト

- ロールバック時のブートストラップ — は、 と呼ばれる Greengrass nucleus 設定パラメータを提供するAWS IoT GreengrassようになりましたBootstrapOnRollback。この機能により、ロールバックデプロイの一部としてブートストラップライフサイクルステップを実行できます。

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.12.0 が利用可能になりました。 新機能 • ロールバックデプロイの一部としてブートストラップライフサイクルステップを実行できます。

リリース: 2023 年 10 月 18 日の AWS IoT Greengrass Core v2.11.3 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.11.3 が提供されます。

リリース日: 2023 年 10 月 18 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.11.3 が利用可能になりました。 バグ修正と機能向上 • nucleus の依存関係が失敗したときにコンポーネントが不適切に開始される可能性がある問題を修正しました。

コンポーネント	詳細
	新機能 • 設定可能な S3 エンドポイントタイプを追加します。
Lambda マネージャー	Lambda マネージャーコンポーネントのバージョン 2.3.1 を利用できます。 バグ修正と機能向上 • 特定のエラーのログレベルを調整します。
ローカル deubg コンソール	Lambda マネージャーコンポーネントのバージョン 2.4.0 を利用できます。 新機能 • ストリームマネージャーのデバッグコンソールを追加します。
ログマネージャー	ログマネージャーコンポーネントのバージョン 2.3.6 を利用できます。 バグ修正と機能向上 • 特定のエラーのログレベルを調整します。
シャドウマネージャー	Shadow Manager コンポーネントのバージョン 2.3.4 を利用できます。 バグ修正と機能向上 • null および空のシャドウ状態ドキュメントのサポートが追加されました。

リリース: 2023 年 8 月 9 日 AWS IoT Greengrass Core v2.11.2 のソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.11.2 が提供されています。

リリース日: 2023 年 8 月 9 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.11.2 が利用可能になりました。 バグ修正と機能向上 • 多数 (50 超) のサブスクリプションが使用されている場合にオフラインとして表示される可能性がある nucleus MQTT 5 クライアントの問題を修正します。 • Docker ダイヤル TCP エラーについての再試行を追加します。

リリース: 2023 年 7 月 21 日 AWS IoT Greengrass Core v2.11.1 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.11.1 が提供されています。

リリース日: 2023 年 7 月 21 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.11.1 が利用可能になりました。 バグ修正と機能向上 • ブートストラップタスクが失敗し、デプロイメタデータファイルが破損している場合に nucleus が起動しない問題を修正します。 • オンデマンド Lambda コンポーネントがデプロイステータスの更新で報告されない問題を修正します。 • 重複する認可ポリシー ID のサポートを追加します。
Lambda マネージャー	Lambda マネージャー のバージョン 2.2.11 が利用可能になりました。

コンポーネント	詳細
	バグ修正と機能向上 • Lambda LegacySubscriptionRouter 設定が変更されても設定が更新されない問題を修正しました。

リリース: 2023 年 6 月 28 日、AWS IoT Greengrass Core v2.11.0 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.11.0 が提供されています。

リリース日: 2023 年 6 月 28 日

リリースハイライト

- 永続ディスクスプーラ — AWS IoT Greengrass では、Greengrass コアデバイスから AWS IoT Core にスプールされたメッセージ用の永続スプーラの実装が可能になりました。このコンポーネントは、これらの送信メッセージをディスクに保存します。詳細については、「ディスクスプーラ」を参照してください。
- ローカルのデプロイの機能向上 — ローカルのデプロイをキャンセルしたり、デプロイの障害処理ポリシーを設定したり、詳細なデプロイステータスを確認したりできるようになりました。
- ロギング速度の向上 — ログマネージャーコンポーネントのログアップロード速度が向上しました。

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデ

バイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.11.0 が利用可能になりました。 新機能 • ローカルのデプロイをキャンセルできます。 • ローカルのデプロイの障害処理ポリシーを設定できます。 • ディスクスプーラプラグインのサポートを追加します。
Greengrass CLI	Greengrass CLI のバージョン 2.11.0 が利用可能になりました。 新機能 • ローカルのデプロイをキャンセルできます。 • ローカルのデプロイの障害処理ポリシーを設定できます。 • 詳細なデプロイステータスのレポートを改善しています。
ディスクスプーラ	ディスクスプーラコンポーネントのバージョン 1.0.0 が利用可能になりました。 • ディスクスプーラコンポーネントは、Greengrass コアデバイスから AWS IoT Core に送信されたメッセージの永続ストレージを提供します。
ログマネージャー	ログマネージャーコンポーネントのバージョン 2.3.5 を利用できます。 改良点 ログのアップロード速度が向上します。

リリース: 2023 年 6 月 21 日、AWS IoT Greengrass Core v2.10.3 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.10.3 が提供されています。

リリース日: 2023 年 6 月 21 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.10.3 が利用可能になりました。 バグ修正と機能向上 • PKCS #11 プロバイダーを使用しているときに Greengrass がデプロイ通知をサブスクライブしない問題を修正しました。

リリース: 2023 年 6 月 5 日、AWS IoT Greengrass Core v2.10.2 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.10.2 が提供されています。

リリース日: 2023 年 6 月 5 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.10.2 が利用可能になりました。 バグ修正と機能向上 • コンポーネントのライフサイクルの大文字と小文字を区別しない解析が可能になります。

コンポーネント	詳細
	• 環境 PATH 変数が正しく再作成されない問題を修正しました。 • 特殊文字を含むユーザー名のストリームマネージャーを含むコンポーネントのプロキシ URI エンコーディングを修正しました。
クライアントデバイス認証	クライアントデバイス認証 コンポーネントのバージョン 2.4.2 が利用可能になりました。 新機能 新しい `startupTimeoutSeconds` 構成オプションを追加します。
Lambda マネージャー	Lambda マネージャー コンポーネントのバージョン 2.2.9 が利用可能になりました。 バグ修正と機能向上 クロックが歪んでいるためにポート番号が壊れている問題を修正します。
ログマネージャー	ログマネージャーコンポーネントのバージョン 2.3.4 を利用できます。 バグ修正と機能向上 • `periodicUploadIntervalSec` パラメータを小数値に設定するためのサポートを追加します。最小値は 1 マイクロ秒です。 • ログマネージャーが`putLogEvents` 制限を守らない問題を修正しました CloudWatch。
MQTT 3.1 ブローカー (モケット)	MQTT 3.1 ブローカー (モケット) コンポーネントのバージョン 2.3.3 が利用可能になりました。 新機能 新しい `startupTimeoutSeconds` 構成オプションを追加します。

コンポーネント	詳細
MQTT ブリッジ	MQTT ブリッジ コンポーネントのバージョン 2.2.6 が利用可能になりました。 新機能 新しい `startupTimeoutSeconds` 構成オプションを追加します。
ストリームマネージャー	ストリームマネージャーコンポーネントのバージョン 2.1.7 が利用可能になりました。 バグ修正と機能向上 ストリームマネージャーがプロキシ構成を正しく読み取れない問題を修正しました。

リリース: 2023 年 5 月 11 日 AWS IoT Greengrass Core v2.10.1 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.10.1 が提供されています。

リリース日: 2023 年 5 月 11 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコア

デバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.10.1 が利用可能になりました。 バグ修正と機能向上 • Jetson Nano など、特定の ARMv8 プロセッサで起動時にクラッシュする可能性がある問題を修正しました。 • Greengrass はコンポーネントの標準入力を閉じないようにしました。これにより、2.10.0 以前の動作に戻ります。
ストリームマネージャー	新しいストリームマネージャーのバージョン 2.1.6 が利用可能になりました。 バグ修正と機能向上 Jetson Nano など、特定の ARMv8 プロセッサで起動時にクラッシュする可能性がある問題を修正しました。

リリース: 2023 年 5 月 9 日 AWS IoT Greengrass Core v2.10.0 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.10.0 を提供するとともに、AWS が提供するコンポーネントの更新を行いました。

リリース日: 2023 年 5 月 9 日

リリースハイライト

- MQTT5 のサポート — AWS IoT Greengrass は MQTT5 を使用して、AWS IoT Core からのメッセージの送受信に対応するようになりました。詳細については、「AWS IoT Core MQTT メッセージの発行」を参照してください。

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.10.0 が利用可能になりました。 新機能 • 空の正規表現に `interpolateComponentConfiguration` サポートを追加します。Greengrass はルート設定オブジェクトから補間するようになりました。

コンポーネント	詳細
	• MQTT5 のサポートを追加します。 • プラグインコンポーネントをスキャンせずに素早く読み込むメカニズムを追加します。 • Greengrass が未使用の Docker イメージを削除することでディスクスペースを節約できるようにします。 バグ修正と機能向上 • ロールバック時に、特定の設定値がデプロイからそのままになる問題を修正します。 • Greengrass nucleus がカスタムの非 AWS 認証情報とデータエンドポイントの AWS ドメインシーケンスを検証する問題を修正します。 • 複数グループの依存関係解決を更新しました。これにより、アクティブなバージョンにロックするのではなく、すべてのグループ依存関係を AWS クラウド ネゴシエーションによって再解決するようになります。この更新により、デプロイエラーコード `INSTALLED_COMPONENT_NOT_FOUND` も削除されます。 • Greengrass nucleus を更新し、既にローカルに存在する Docker イメージのダウンロードをスキップするようにしました。 • Greengrass nucleus を更新し、タイムアウトが切れる前にコンポーネントのインストールステップを再開するようにしました。 • 追加のマイナー修正と機能向上。
シャドウマネージャー	新しいシャドウマネージャーのバージョン 2.3.2 が利用可能になりました。 バグ修正と機能向上 ローカルシャドウデータベースが破損している場合に、シャドウマネージャーが `BROKEN` 状態になる問題を修正します。

リリース: 2023 年 4 月 20 日 AWS IoT Greengrass Core v2.9.6 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.6 が提供されています。

リリース日: 2023 年 4 月 20 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.9.6 が利用可能になりました。 バグ修正と機能向上 • Greengrass のデプロイがエラー LAUNCH_DIRECTORY_CORRUPTED で失敗し、その後のデバイス再起動で Greengrass の起動に失敗する問題を修正しました。このエラーは、Greengrass の再起動が必要なデプロイで、複数のモノグループ間で Greengrass デバイスを移動した場合に発生する場合があります。

リリース: 2023 年 3 月 30 日 AWS IoT Greengrass Core v2.9.5 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.5 が提供されています。

リリース日: 2023 年 3 月 30 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.9.5 が利用可能になりました。 新機能 • Greengrass nucleus ソフトウェアの署名検証をサポートするようになりました。

コンポーネント	詳細
	バグ修正と機能向上 • ローカル recipe のメタデータ領域が Greengrass nucleus の起動領域と一致しない場合に、デプロイが失敗する問題を修正しました。Greengrass nucleus は、このような場合にクラウドと再交渉するようになりました。 • MQTT メッセージスプーラが一杯になり、メッセージが削除されない問題を修正しました。 • 追加のマイナー修正と機能向上。

リリース: 2023 年 2 月 24 日 AWS IoT Greengrass Core v2.9.4 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.4 が提供されています。

リリース日: 2023 年 2 月 24 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS

IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.9.4 が利用可能になりました。 バグ修正と機能向上 • QOS 0 メッセージを破棄する前に null メッセージの有無がチェックされます。 • ジョブステータスの詳細値は、1024 文字の制限を超えると、切り捨てられます。 • Windows のブートストラップスクリプトを更新して、パスにスペースが含まれている場合に Greengrass ルートパスを正しく読み取れるようにします。 • サブスクリプションの応答が送信されなかった場合にクライアントメッセージが破棄されるように、AWS IoT Core のサブスクライブを更新します。 • メイン設定ファイルが壊れているか見つからない場合に、nucleus がバックアップファイルから設定を読み込むようにします。

リリース: 2023 年 2 月 1 日 AWS IoT Greengrass Core v2.9.3 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.3 が提供されています。

リリース日: 2023 年 2 月 1 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.9.3 が利用可能になりました。 バグ修正と機能向上 • MQTT クライアント ID が重複しないようにします。 • ファイルの読み取りと書き込みがより堅牢になり、破損の回避と回復を図っています。 • ネットワークに関連する特定のエラーの発生時に、Docker イメージのプルを再試行するようになりました。 • MQTT 接続の `noProxyAddresses` オプションを追加します。

リリース: 2022 年 12 月 22 日 AWS IoT Greengrass Core v2.9.2 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.2 が提供されています。

リリース日: 2022 年 12 月 22 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.9.2 が利用可能になりました。 バグ修正と機能向上 • 進行中のデプロイに `interpolateComponentConfiguration` の設定が適用されない問題を修正しました。 • OSHI を使用して、すべての子プロセスを一覧表示します。

リリース: 2022 年 11 月 18 日 AWS IoT Greengrass Core v2.9.1 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.1 を提供するとともに、AWS から提供されるコンポーネントの更新を行いました。

リリース日: 2022 年 11 月 18 日

リリースハイライト

- ログマネージャー — ログマネージャーは、新しいファイルがローテーションされるのを待つことなく、アクティブなログファイルを処理して直接アップロードするようになりました。この改善により、ログの遅延が大幅に軽減されます。詳細については、「ログマネージャー」を参照してください。

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.9.1 が利用可能になりました。 バグ修正と機能向上 • デプロイがプラグインコンポーネントを削除する場合に Greengrass が再起動する問題の修正を追加しました。
ログマネージャー	新しいログマネージャーのバージョン 2.3.0 が利用可能になりました。 **Note** ログマネージャー 2.3.0 にアップグレードする場合は、Greengrass nucleus 2.9.1 にアップグレードすることをお勧めします。 新機能 • 新しいファイルがローテーションされるのを待つことなく、アクティブなログファイルを処理して直接アップロードすることによって、ログの遅延を軽減します。 バグ修正と機能向上 • 一意の名前を持つファイルをローテーションする際のログローテーションのサポートを改善しました。 • 追加のマイナー修正と機能向上。

リリース: 2022 年 11 月 15 日 AWS IoT Greengrass Core v2.9.0 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.0 を提供するとともに、AWS から提供されるコンポーネントの更新を行いました。

リリース日: 2022 年 11 月 15 日

リリースハイライト

- オフライン認証 – AWS IoT Greengrass でオフライン認証がサポートされるようになりました。コアデバイスがクラウドに接続されていない場合でも、クライアントデバイスがコアデバイスに接続できるように、AWS IoT Greengrass コアデバイスを設定できます。詳細については、「Offline authentication」(オフライン認証) を参照してください。
- サブデプロイ — サブデプロイを作成できるようになりました。サブデプロイを使用して、失敗したデプロイを解決できます。サブデプロイごとに、失敗したデプロイの異なる設定を、デバイスのより小さなサブセットでテストできます。詳細については、「サブデプロイを作成する」を参照してください。

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.9.0 が利用可能になりました。 新機能 • サブデプロイを作成して、デバイスのより小さなサブセットでデプロイを再試行する機能を追加しました。この機能により、失敗したデプロイをより効率的にテストして解決できます。 バグ修正と機能向上 • useradd、groupadd、および usermod を持たないシステムのサポートを改善しました。 • 追加のマイナー修正と機能向上。
クライアントデバイス認証	クライアントデバイス認証コンポーネントのバージョン 2.3.0 が利用可能になりました。 新機能 • クライアントデバイスのオフライン認証のサポートを追加しました。この機能により、コアデバイスがインターネットに接続されていない場合でも、クライアントデバイスは引き続きコアデバイスに接続できます。 • お客様が提供する認証機関 (CA) のサポートを追加しました。コアデバイスは、お客様が提供する CA をルート証明書として使用して MQTT ブローカー証明書を生成します。
MQTT 5 ブローカー (EMQX)	MQTT 5 ブローカー (EMQX) コンポーネントのバージョン 1.2.0 が利用可能になりました。 新機能 証明書チェーンのサポートを追加しました。
モケット MQTT ブローカー	Moquette MQTT ブローカーコンポーネントの新バージョン 2.3.0 が利用可能になりました。 新機能 証明書チェーンのサポートを追加しました。

コンポーネント	詳細
シークレットマネージャー	新しいシークレットマネージャーのバージョン 2.1.4 が利用可能になりました。 バグ修正と機能向上 シークレットマネージャーがデプロイされ、Greengrass nucleus が再起動されたときに、キャッシュされたシークレットが削除される問題を修正しました。
ストリームマネージャー	新しいストリームマネージャーのバージョン 2.1.2 が利用可能になりました。 バグ修正と機能向上 英語以外の言語を使用する Windows OS における問題を修正しました。

リリース: 2022 年 10 月 13 日、AWS IoT Greengrass Core v2.8.1 ソフトウェアの更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.8.1 が提供されています。

リリース日: 2022 年 10 月 13 日

> **Note**
>
> Greengrass nucleus バージョン 2.8.0 を使用している場合は、Greengrass nucleus バージョン 2.8.1 にアップグレードすることを強くお勧めします。

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.8.1 が利用可能になりました。 バグ修正と機能向上 • Greengrass API エラーからデプロイエラーコードが正しく生成されなかった問題を修正します。 • デプロイ中にコンポーネントが ERRORED 状態に達したときに、フリートステータスの更新によって不正確な情報が送信される問題を修正します。 • Greengrass の既存のサブスクリプションが 50 を超える場合にデプロイを完了できなかった問題を修正します。

リリース: 2022 年 10 月 7 日、AWS IoT Greengrass Core v2.8.0 ソフトウェアの更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.8.0 と、MQTT 5 ブローカー (EMQX) コンポーネントのバージョン 1.1.0 が提供されています。

リリース日: 2022 年 10 月 7 日

リリースハイライト

- デプロイのエラーコード – Greengrass nucleus は、コンポーネントのデプロイを完了できない場合に、デプロイのヘルスステータスレスポンス (詳細なエラーコードを含む) を報告するようになりました。詳細については、「詳細なデプロイエラーコード」を参照してください。
- コンポーネントのエラーステータス – Greengrass nucleus は、コンポーネントが BROKEN または ERRORED 状態になったときに、コンポーネントのヘルスステータスレスポンス (詳細なエラーコードを含む) を報告するようになりました。詳細については、「詳細なコンポーネントのステータスコード」を参照してください。

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.8.0 が利用可能になりました。

コンポーネント	詳細
	新機能 • コアデバイスへのコンポーネントのデプロイで問題が発生した場合に、デプロイのヘルスステータスのレスポンス (詳細なエラーコードを含む) を報告するように、Greengrass nucleus を更新します。詳細については、「詳細なデプロイエラーコード」を参照してください。 • コンポーネントが BROKEN または ERRORED の状態になったときに、コンポーネントのヘルスステータスレスポンス (詳細なエラーコードを含む) を報告するように、Greengrass nucleus を更新します。詳細については、「詳細なコンポーネントのステータスコード」を参照してください。 • ステータスメッセージフィールドを展開して、デバイスのクラウド可用性情報を改善します。 • フリートステータスサービスの堅牢性を向上させます。 バグ修正と機能向上 • 設定が変更されたときに、壊れたコンポーネントを再インストールできるようにします。 • ブートストラップデプロイ中に nucleus が再起動するとデプロイが失敗する問題を修正します。 • ルートパスにスペースが含まれているとインストールが失敗する Windows の問題を修正します。 • デプロイ中にシャットダウンされたコンポーネントが新しいバージョンのシャットダウンスクリプトを使用する問題を修正します。 • シャットダウンのさまざまな改善。 • 追加のマイナー修正と機能向上。

コンポーネント	詳細
MQTT 5 ブローカー (EMQX)	MQTT 5 ブローカー (EMQX) コンポーネントのバージョン 1.1.0 を利用できます。 新機能 • ブローカーオプションやプラグインを含む EMQX 設定のサポートを追加します。 バグ修正と機能向上 • EMQX をバージョン 4.4.9 に更新します。

リリース: 2022 年 7 月 28 日 AWS IoT Greengrass Core v2.7.0 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.7.0、ストリームマネージャーコンポーネントのバージョン 2.1.0、および Lambda マネージャーコンポーネントのバージョン 2.2.5 が提供されます。

リリース日: 2022 年 7 月 28 日

リリースハイライト

- ストリームマネージャーのテレメトリメトリクス – ストリームマネージャーはテレメトリメトリクスを Amazon に自動的に送信するようになりました。これにより EventBridge、コアデバイスがアップロードするデータ量をモニタリングおよび分析するクラウドアプリケーションを作成できます。詳細については、「AWS IoT Greengrass コアデバイスからシステムヘルステレメトリデータを収集する」を参照してください。
- カスタム認証機関 (CA) – CA が AWS IoT に登録されていないカスタム証明書によって署名された、クライアント証明書がサポート対象になりました。詳細については、「プライベート CA によって署名されたデバイス証明書を使用する」を参照してください。

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.7.0 が利用可能になりました。 新機能 • コアデバイスがローカルでデプロイを適用する際、更新されたステータスが AWS IoT Greengrass のクラウドに送信されるように、Greengrass nucleus を更新しました。 • CA が AWS IoT に登録されていない場合に、カスタム認証局 (CA) によって署名されたクライアント証明書を使用できるようにしました。この機能を使用するには、新しい `greengrassDataPlaneEndpoint` 設定オプションに `iotdata` を設定します。詳細については、「プライベート CA によって署名されたデバイス証明書を使用する」を参照してください。

コンポーネント	詳細
	バグ修正と機能向上 • Nucleus が停止または再起動された場合の特定のシナリオで、Greengrass nucleus がデプロイをロールバックする問題を修正しました。今後 Nucleus は、再起動した後にも同じデプロイで動作します。 • ソフトウェアをシステムサービスとしてセットアップする際、`--start` 引数が反映されるように Greengrass インストーラを更新しました。 • nucleus がコンポーネントを更新したイベントのデプロイ ID を設定するように、SubscribeToComponentUpdates の動作を更新しました。 • 追加のマイナー修正と機能向上。
ストリームマネージャー	ストリームマネージャーコンポーネントのバージョン 2.1.0 が利用可能になりました。 新機能 • このコンポーネントを更新して、テレメトリメトリクスを Amazon に自動的に送信します EventBridge。詳細については、「AWS IoT Greengrass コアデバイスからシステムヘルステレメトリデータを収集する」を参照してください。 この機能を使用するには、Greengrass nucleus コンポーネントの v2.7.0 以降が必要です。 • Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。

コンポーネント	詳細
Lambda マネージャー	Lambda マネージャーコンポーネントのバージョン 2.2.5 が利用可能になりました。 新機能 • ローカルの公開/サブスクライブメッセージにサブスクライブするイベントソースで、MQTT トピックのワイルドカードが使用できるようになりました。 この機能を使用するには、Greengrass nucleus コンポーネントの v2.6.0 以降が必要です。 • Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。

リリース: 2022 年 6 月 27 日、AWS IoT Greengrass Core v2.6.0 ソフトウェアの更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.6.0 の提供と、新しい AWS 提供コンポーネントの追加、さらに AWS から提供されているコンポーネントの更新が行われています

リリース日: 2022 年 6 月 27 日

リリースハイライト

- ローカルの公開/サブスクライブトピックでのワイルドカード – ローカルの公開/サブスクライブトピックにサブスクライブする際、MQTT ワイルドカードの使用が可能になりました。詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする と SubscribeToTopic」を参照してください。
- クライアントのデバイスシャドウのサポート – カスタムコンポーネントからクライアントのデバイスシャドウを操作し、そのデバイスシャドウを AWS IoT Core と同期できるようになりました。詳細については、「クライアントデバイスシャドウとやり取りして同期する」を参照してください。
- クライアントデバイスでのローカル MQTT 5 のサポート – クライアントデバイスとコアデバイス間の通信で MQTT 5 機能を使用するために、EMQX MQTT 5 ブローカーをデプロイできるように

なりました。詳細については、「MQTT 5 ブローカー (EMQX) と クライアントデバイスをコアデバイスに接続する」を参照してください。

- コンポーネント設定のレシピ変数 – コンポーネント設定で特定のレシピ変数を使用できるようになりました。これらのレシピ変数は、コンポーネントのデフォルト設定を定義する際、またはデプロイでコンポーネントを設定する際に、レシピとして使用できます。詳細については、「レシピ変数 と マージ更新で recipe 変数を使用する」を参照してください。
- IPC 認証ポリシー内のワイルドカード – * ワイルドカードを使用して、プロセス間通信 (IPC) 認証ポリシーの中で任意の文字の組み合わせに一致させることが可能になりました。このワイルドカードを使用すると、単一の認証ポリシーで複数のリソースに対するアクセスを許可できます。詳細については、「承認ポリシー内のワイルドカード」を参照してください。
- ローカルでのデプロイとコンポーネントを管理する IPC オペレーション – ローカルのデプロイを管理し、コンポーネントの詳細を表示する、カスタムコンポーネントを開発できるようになりました。詳細については、「IPC: Manage local deployments and components」(IPC: ローカルのデプロイとコンポーネントを管理する) を参照してください。
- クライアントデバイスを認証および承認する IPC オペレーション – これらのオペレーションを使用して、カスタムのローカルブローカーコンポーネントを作成できるようになりました。詳細については、「IPC: Authenticate and authorize client devices」(IPC: クライアントデバイスの認証と承認) を参照してください。

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。

デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.6.0 が利用可能になりました。 新機能 • ローカルで公開/サブスクライブされるトピックをサブスクライブする際に、MQTT ワイルドカードが使用できるようになりました。詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする と SubscribeToTopic」を参照してください。 • コンポーネント構成で、*component_dependency_name* :configuration: *json_pointer* recipe 変数以外の recipe 変数がサポートされました。これらの recipe 変数は、recipe でコンポーネントの DefaultConfiguration を定義している場合、またはデプロイでコンポーネントを構成している場合に使用できます。この機能を有効にするには、interpolateComponentConfiguration設定オプションを に設定しますtrue。詳細については、「レシピ変数 とマージ更新で recipe 変数を使用する」を参照してください。 • プロセス間通信 (IPC) 承認ポリシーでの、* ワイルドカードの完全なサポートを追加しました。これにより、リソース文字列内の * 文字を、文字の任意の組み合わせと一致させる指定が可能になりました。詳細については、「承認ポリシー内のワイルドカード」を参照してください。 • Greengrass CLI が使用する IPC オペレーションを呼び出すために、カスタムコンポーネントが使用できるようになりました。これらの IPC オペレーションにより、ローカルでのデプロイ管理、コンポーネントの詳細の表示、およびローカルのデバッグコンソールへのサインイン用のパスワード生成を行うことができます。詳細については、「IPC: Manage local deployments and components」(IPC: ローカルのデプロイとコンポーネントを管理する) を参照してください。

コンポーネント	詳細
	バグ修正と機能向上 • 特定のシナリオで、ハード依存関係が再起動もしくはステータスを変更をした際に、依存関係のあるコンポーネントが反応しない問題を修正しました。 • デプロイが失敗した場合に、コアデバイスが AWS IoT Greengrass のクラウドサービスに報告するエラーメッセージの改善を行いました。 • 特定のシナリオでの Greengrass nucleus の再起動時、nucleus がモノのデプロイを 2 回適用していた問題を修正しました。 • 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。
MQTT 5 ブローカー (EMQX)	新しい EMQX MQTT 5 ブローカーコンポーネントのバージョン v 1.0.0 が利用可能になりました。 新機能 • ローカル EMQX MQTT 5 ブローカーのサポートが追加されました。クライアントデバイスはこの MQTT ブローカーに接続し、MQTT 5 の機能を使用するコアデバイスと通信できます。

コンポーネント	詳細
シャドウマネージャー	シャドウマネージャーコンポーネントのバージョン 2.2.0 が利用可能になりました。 新機能 • ローカルの公開/サブスクライブインターフェイスを通じた、ローカルシャドウサービスに関するサポートが追加されています。シャドウ MQTT トピックで、ローカルの公開/サブスクライブのメッセージブローカーと通信し、コアデバイスのシャドウを取得、更新、削除することが可能です。この機能により、MQTT ブリッジを使用してクライアントデバイスをローカルシャドウサービスに接続することで、シャドウトピックのメッセージを、クライアントデバイスとローカルの公開/サブスクライブインターフェイス間でリレーすることが可能になります。 この機能には、Greengrass nucleus コンポーネントの v2.6.0 以降が必要です。クライアントデバイスをローカルシャドウサービスに接続するには、MQTT ブリッジコンポーネント の v2.2.0 以降を使用する必要があります。 • ローカルシャドウサービスと AWS クラウド の間で、シャドウを同期する方向をカスタマイズするために、`direction` オプションを設定できるようになりました。このオプションを設定すると、AWS クラウドへの帯域幅と接続数を低減できます。
クライアントデバイス認証	クライアントデバイス認証コンポーネントのバージョン 2.2.0 が利用可能になりました。 新機能 • プロセス間通信 (IPC) オペレーションを呼び出して、クライアントデバイスを認証および承認する、カスタムコンポーネントのサポートを追加しました。例えば、カスタム MQTT ブローカーコンポーネントで、これらのオペレーションを使用できます。詳細については、「IPC: Authenticate and authorize client devices」(IPC: クライアントデバイスの認証と承認) を参照してください。 • このコンポーネントの処理内容を調整するための、`maxActiveAuthTokens` 、`cloudQueueSize` 、および `threadPoolSize` オプションを追加しました。

コンポーネント	詳細
MQTT ブリッジ	MQTT ブリッジコンポーネントのバージョン 2.2.0 が利用可能になりました。 新機能 • ローカルの公開/サブスクライブをソースメッセージブローカーとして指定する場合の、MQTT トピックのワイルドカード (# および +) に関するサポートが追加されています。 この機能を使用するには、Greengrass nucleus コンポーネントの v2.6.0 以降が必要です。 • ターゲットトピックがメッセージを伝達するためのプレフィックスを追加するように MQTT ブリッジを設定するための、`targetTopicPrefix` オプションが追加されています。
Greengrass CLI	Greengrass CLI のバージョン 2.6.0 が利用可能になりました。 新機能 • Greengrass CLI が使用するプロセス間通信 (IPC) のオペレーションを呼び出す、カスタムコンポーネントに関するサポートが追加されました。これらの IPC オペレーションにより、ローカルでのデプロイ管理、コンポーネントの詳細の表示、およびローカルのデバッグコンソールへのサインイン用のパスワード生成を行うことができます。詳細については、「IPC: Manage local deployments and components」(IPC: ローカルのデプロイとコンポーネントを管理する) を参照してください。 バグ修正と機能向上 • 追加のマイナー修正と機能向上。

リリース: 2022 年 5 月 31 日 AWS IoT Greengrass Core v2.5.6 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.5.6 と、ログマネージャーコンポーネントのバージョン 2.2.4 が提供されています。

リリース日: 2022 年 5 月 31 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.5.6 を利用できます。 新機能 • ECC キーを使用するハードウェアセキュリティモジュールのサポートが追加されました。ハードウェアセキュリティモジュール (HSM) を使用して、デバイスのプライベートキーと証明書を安全に保存できます。詳細については、「ハードウェアセキュリティ統合」を参照してください。

コンポーネント	詳細
	バグ修正と機能向上 • 特定のシナリオで壊れたインストールスクリプトを使用してコンポーネントをデプロイすると、デプロイが完了しない問題を修正しました。 • 起動時のパフォーマンスが向上しました。 • 追加のマイナー修正と機能向上。
ログマネージャー	ログマネージャーコンポーネントのバージョン 2.2.4 を利用できます。 バグ修正と機能向上 • 無効な設定を処理するときの安定性を向上させます。 • 追加のマイナー修正と機能向上。

リリース: 2022 年 4 月 6 日 AWS IoT Greengrass Core v2.5.5 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.5.5 が提供されています。

リリース日: 2022 年 4 月 6 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコア

デバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.5.5 を利用できます。 新機能 • コンポーネントの `GG_ROOT_CA_PATH` 環境変数が追加され、カスタムコンポーネントのルート認証局 (CA) 証明書にアクセスできるようになりました。 バグ修正と機能向上 • 英語以外の表示言語を使用する Windows デバイスのサポートが追加されました。 • Greengrass nucleus がインストーラ引数のブール値を解析する方法が更新され、`true` 値を指定するブール値なしでブール引数を指定できるようになりました。たとえば、`--provision true` の代わりに `--provision` をクリックして、自動リソースプロビジョニングを使用してインストールできるようになりました。 • 特定のシナリオでプロビジョニングした後に、コアデバイスがステータスを AWS IoT Greengrass クラウドサービスに報告しない問題を修正しました。 • 追加のマイナー修正と機能向上。

リリース: 2022 年 3 月 23 日 AWS IoT Greengrass Core v2.5.4 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.5.4 と Lambda ランチャーコンポーネントのバージョン 2.0.10 が提供されています。

リリース日: 2022 年 3 月 23 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.5.4 を利用できます。 バグ修正と機能向上 • 一般的なバグ修正と機能強化。
Lambda ランチャー	Lambda ランチャーコンポーネントのバージョン 2.0.10 を利用できます。 バグ修正と機能向上 • 一般的なバグ修正と機能強化。

リリース: 2022 年 1 月 6 日 AWS IoT Greengrass Core v2.5.3 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントと新しい PKCS#11 プロバイダコンポーネントのバージョン 2.5.3 を提供します。

リリース日: 2022 年 1 月 6 日

リリースハイライト

- ハードウェアセキュリティ統合 - AWS IoT Greengrass Core ソフトウェアをハードウェアセキュリティモジュール (HSM) に安全に保管するプライベートキーと証明書を使用するように設定できるようになります。詳細については、「ハードウェアセキュリティ統合」を参照してください。

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.5.3 を利用できます。 新機能 • ハードウェアセキュリティ統合のサポートが追加されました。ハードウェアセキュリティモジュール (HSM) を使用して、デバイスのプライベートキーと証明書を安全に保存できます。詳細については、「ハードウェアセキュリティ統合」を参照してください。 バグ修正と機能向上 • nucleus が AWS IoT Core と MQTT 接続を確立している際のランタイム例外に関する問題を修正しました。
PKCS#11 プロバイダ	PKCS#11 プロバイダコンポーネントのバージョン 2.0.0 を利用できます。 新機能 • ハードウェアセキュリティ統合のサポートが追加されました。ハードウェアセキュリティモジュール (HSM) を使用して、デバイスのプライベートキーと証明書を安全に保存できます。詳細については、「ハードウェアセキュリティ統合」を参照してください。

リリース: 2021 年 12 月 3 日 AWS IoT Greengrass Core v2.5.2 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.5.2 が提供されています。

リリース日: 2021 年 12 月 3 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.5.2 を利用できます。 バグ修正と機能向上 • Greengrass nucleus の更新後、デバイスを停止または再起動した後、Windows サービスが再び起動しない問題を修正しました。
AWS IoT Device Defender	AWS IoT Device Defenderコンポーネントのバージョン 3.0.1 を利用できます。 AWS IoT Device Defenderコンポーネントのこのバージョンでは、バージョン 2.x とは異なる設定パラメータが必要です。バージョン 2.x でデフォルト以外の設定を使用し、v2.x から v3.x にアップグレードする場合は、コンポーネントの設定を更新する必要があります。設定ファイルの使用の詳細については、「AWS IoT Device Defender コンポーネントの設定」を参照してください。 新機能 • Windows を実行するコアデバイスのサポートが追加されました。 • コンポーネントタイプを Lambda コンポーネントから汎用コンポーネントに変更しました。このコンポーネントは、サブスクリプションを作成

コンポーネント	詳細
	するのにレガシーサブスクリプションルーターコンポーネントに依存しなくなりました。 • コンポーネントの依存関係をインストールするインストールスクリプトをオプションで無効にできる、新しい `UseInstaller` 設定パラメータが追加されました。

リリース: 2021 年 11 月 23 日 AWS IoT Greengrass Core v2.5.1 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.5.1 が提供されています。

リリース日: 2021 年 11 月 23 日

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.5.1 を利用できます。 バグ修正と機能向上 • Windows の 32 ビットバージョンの Java Runtime Environment (JRE) のサポートが追加されました。 • AWS IoT ポリシーで、`greengrass:ListThingGroupsForCoreDevice` 権限が付与されないコアデバイスのモノグループの削除動作が変更されました。このバージョンでは、デプロイは続行され、警告が記録され、モノグループからコアデバイスを削除してもコンポーネントは削除されません。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。 • Greengrass nucleus が Greengrass コンポーネントプロセスで利用できるようにするシステム環境変数の問題を修正しました。これで、コンポーネントを再起動して、最新のシステム環境変数を使用できます。

リリース: 2021 年 11 月 12 日 AWS IoT Greengrass Core v2.5.0 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.5.0 と AWS から新たに提供されるコンポーネントおよび AWS から提供されるコンポーネントの更新を提供します。

リリース日: 2021 年 11 月 12 日

リリースハイライト

- Windows デバイスのサポート - Windows オペレーティングシステムを実行しているデバイス上で AWS IoT Greengrass Core ソフトウェアを実行できるようになりました。詳細については、「サポートされているプラットフォームと要件 と オペレーティングシステム別 Greengrass 機能の互換性」を参照してください。
- 新しいモノグループの削除動作 - モノグループからコアデバイスを削除すると、そのデバイスへの次のデプロイでそのモノグループのコンポーネントを削除できるようになりました。

> **Important**
>
> この変更の結果、コアデバイスの AWS IoT ポリシーには `greengrass:ListThingGroupsForCoreDevice` 権限が必要になります。リソースをプロビジョニングするために AWS IoT Greengrass Core ソフトウェアインストーラを使用する場合、デフォルトの AWS IoT ポリシーは `greengrass:*` を許可します。これには、この権限が含まれています。詳細については、「AWS IoT Greengrass のデバイス認証と認可」を参照してください。

- ハードウェアセキュリティサポート - AWS IoT Greengrass Core ソフトウェアでハードウェアセキュリティモジュール (HSM) を使用するように設定できるようになり、デバイスのプライベートキーと証明書を安全に保存できるようになりました。詳細については、「ハードウェアセキュリティ統合」を参照してください。
- HTTPS プロキシサポート - AWS IoT Greengrass Core ソフトウェアを HTTPS プロキシ経由で接続するように設定できるようになりました。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

リリース詳細

- プラットフォームサポートの更新
- パブリックコンポーネントの更新

プラットフォームサポートの更新

プラットフォーム	詳細
Windows	AWS IoT Greengrass に、次のバージョンの Windows で AWS IoT Greengrass Core ソフトウェアを実行するためのサポートが追加されました。 • Windows 10 • [Windows Server 2019] 詳細については、「サポートされているプラットフォームと要件 と オペレーティングシステム別 Greengrass 機能の互換性」を参照してください。

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.5.0 を利用できます。 新機能 • Windows を実行するコアデバイスのサポートが追加されました。 • モノグループを削除する動作が変更されました。このバージョンでは、モノグループからコアデバイスを削除して、次のデプロイでそのモノグループのコンポーネントをアンインストールできるようになりました。 この変更の結果、コアデバイスの AWS IoT ポリシーには `greengrass:ListThingGroupsForCoreDevice` 権限が必要になります。リソースをプロビジョニングするために AWS IoT Greengrass Core ソフトウェアインストーラを使用する場合、デフォルトの AWS IoT ポリシーは `greengrass:*` を許可します。これには、この権限が含まれています。詳細については、「AWS IoT Greengrass のデバイス認証と認可」を参照してください。

コンポーネント	詳細
	• HTTPS プロキシ設定のサポートが追加されました。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。 • 新しい `windowsUser` 設定パラメータが追加されました。このパラメータを使用して、Windows コアデバイスでコンポーネントを実行するために使用するデフォルトユーザーを指定できます。詳細については、「コンポーネントを実行するユーザーを設定する」を参照してください。 • HTTP リクエストのタイムアウトをカスタマイズして、低速ネットワークでのパフォーマンスを向上させるために使用できる新しい `httpClient` 設定オプションが追加されました。詳細については、httpClient 設定パラメータを参照してください。 バグ修正と機能向上 • コンポーネントからコアデバイスを再起動するためのブートストラップライフサイクルオプションが修正されました。 • recipe 変数でハイフンがサポートされるようになりました。 • オンデマンド Lambda 関数コンポーネントの IPC 認証が修正されました。 • ログメッセージを改善し、クリティカルでないログを `INFO` から `DEBUG` レベルに変更したことで、ログの利便性が高まりました。 • 自動プロビジョニング機能を備えた AWS IoT Greengrass Core ソフトウェアをインストールする際に Greengrass nucleus が作成するトークン交換ロールから `iot:DescribeCertificate` 権限を取り除きました。この権限は Greengrass nucleus では使用されません。 • `iam:CreatePolicy` が、同じポリシーで利用できる場合、自動プロビジョニングスクリプトで `iam:GetPolicy` 権限が必要ないように問題を修正しました。 • 追加のマイナー修正と機能向上。

コンポーネント	詳細
Greengrass CLI	Greengrass CLI のバージョン 2.5.0 を利用できます。 新機能 • Windows を実行するコアデバイスのサポートが追加されました。 • Windows デバイスで Greengrass CLI を使用するために、システムグループを認可するために指定できる新しい `AuthorizedWindowsGroups` 設定パラメータを追加しました。 • ローカルデプロイ用の `windowsUser` パラメータが追加されました。このパラメータを使用して、Windows コアデバイスでコンポーネントを実行するために使用するユーザーを指定できます。

コンポーネント	詳細
CloudWatch メトリクス	CloudWatch メトリクスコンポーネントのバージョン 3.0.0 を利用できます。 このバージョンの CloudWatch メトリクスコンポーネントでは、バージョン 2.x とは異なる設定パラメータが必要です。バージョン 2.x でデフォルト以外の設定を使用し、v2.x から v3.x にアップグレードする場合は、コンポーネントの設定を更新する必要があります。詳細については、「CloudWatchメトリクスコンポーネントの設定」を参照してください。 新機能 • Windows を実行するコアデバイスのサポートが追加されました。 • コンポーネントタイプを Lambda コンポーネントから汎用コンポーネントに変更しました。このコンポーネントは、サブスクリプションを作成するのにレガシーサブスクリプションルーターコンポーネントに依存しなくなりました。 • コンポーネントがメッセージを受信するためにサブスクライブするトピックを指定するための、新しい `InputTopic` 設定パラメータが追加されました。 • コンポーネントがステータスレスポンスをパブリッシュするトピックを指定するための、新しい `OutputTopic` 設定パラメータが追加されました。 • AWS IoT Core MQTT トピックのパブリッシュとサブスクライブを行うかどうかを指定するための新しい `PubSubToIoTCore` 設定パラメータが追加されました。 • コンポーネントの依存関係をインストールするインストールスクリプトをオプションで無効にできる、新しい `UseInstaller` 設定パラメータが追加されました。 バグ修正と機能向上 入力データでの重複するタイムスタンプへのサポートが追加されました。

コンポーネント	詳細
Lambda マネージャー	Lambda マネージャーコンポーネントのバージョン 2.2.0 を利用できます。 バグ修正と機能向上 • 再起動後に Lambda 関数がログを書き込めない問題を修正しました。 • トピックにワイルドカードがある場合、レガシーサブスクリプションルータが重複するメッセージを送信する問題を修正しました。 • ピン留めされていない Lambda 関数が AWS IoT Device SDK で Greengrass プロセス間通信 (IPC) ライブラリを使用できない問題を修正しました。

リリース: 2021 年 8 月 3 日 AWS IoT Greengrass Core v2.4.0 のソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.4.0 と AWS から新たに提供されるコンポーネントおよび AWS から提供されるコンポーネントの更新を提供します。

リリース日: 2021 年 8 月 3 日

リリースハイライト

- システムリソースの制限 - Greengrass nucleus コンポーネントでシステムリソース制限がサポートされるようになりました。各コンポーネントのプロセスがコアデバイスで使用できる CPU および RAM の最大使用数を設定できます。詳細については、「コンポーネントのシステムリソース制限を設定する」を参照してください。
- コンポーネントの一時停止/再開 - Greengrass nucleus で、コンポーネントの一時停止と再開がサポートされるようになりました。プロセス間通信 (IPC) ライブラリを使用して、他のコンポーネントのプロセスを一時停止および再開するカスタムコンポーネントを開発できます。詳細については、「PauseComponent と ResumeComponent」を参照してください。
- AWS IoT フリートプロビジョニングでのインストール - 新しい AWS IoT フリートプロビジョニングプラグインを使用して、AWS IoT に接続するデバイスの AWS IoT Greengrass Core ソフトウェアをインストールして、必要な AWS リソースをプロビジョニングします。デバイスは、クレーム証明書を使用してプロビジョニングします。製造中にクレーム証明書をデバイスに埋め込むことができ、各デバイスがオンラインになるとすぐにプロビジョニングできます。詳細については、

「AWS IoT フリートプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。

- カスタムプロビジョニングによるインストール - デバイスに AWS IoT Greengrass Core ソフトウェアをインストールする際に、必要な AWS リソースをプロビジョンするためのカスタムプロビジョニングプラグインを開発します。インストール中に実行される Java アプリケーションを作成して、カスタムユースケース用の Greengrass コアデバイスをセットアップできます。詳細については、「カスタムリソースプロビジョニングを使用して AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.4.0 を利用できます。

コンポーネント	詳細
	新機能 • システムリソース制限のサポートを追加します。各コンポーネントのプロセスがコアデバイスで使用できる CPU および RAM の最大使用数を設定できます。詳細については、「コンポーネントのシステムリソース制限を設定する」を参照してください。 • コンポーネントを一時停止および再開するための IPC 操作が追加されました。詳細については、「PauseComponent と ResumeComponent」を参照してください。 • プラグインのプロビジョニングのサポートが追加されました。インストール中に実行する JAR ファイルを指定して、Greengrass コアデバイスの必要な AWS リソースをプロビジョニングできます。Greengrass nucleus には、カスタムプロビジョニングプラグインを開発するために実装できるインターフェースが含まれています。詳細については、「カスタムリソースプロビジョニングを使用して AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。 • AWS IoT Greengrass Core ソフトウェアのインストーラにオプションの `thing-name-policy` 引数が追加されました。このオプションを使用すると、自動リソースプロビジョニング機能を備えた AWS IoT Greengrass Core ソフトウェアをインストールする際に、既存またはカスタムの AWS IoT ポリシーを指定できます。 バグ修正と機能向上 • 起動時にログ設定を更新します。これにより、起動時にログ設定が適用されなかった問題が修正されます。 • インストール中に、Greengrass ルートフォルダ内のコンポーネントストアを指すように nucleus ローダーのシンボリックリンクを更新します。この更新により、AWS IoT Greengrass Core ソフトウェアインストール時にダウンロードした JAR ファイルおよびその他の nucleus アーティファクトを削除できるようになりました。 • 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。

コンポーネント	詳細
Greengrass CLI	Greengrass CLI のバージョン 2.4.0 を利用できます。 新機能 • システムリソース制限のサポートを追加します。ローカルデプロイを作成するときに、各コンポーネントのプロセスがコアデバイスで使用できる CPU および RAM の最大使用数を設定できます。詳細については、「コンポーネントのシステムリソース制限を設定する」と「デプロイ作成コマンド」を参照してください。
クレームによる AWS IoT フリートプロビジョニング	クレームプラグインによる AWS IoT フリートプロビジョニングが利用可能になりました。詳細については、「AWS IoT フリートプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。 新機能 • AWS IoT フリートプロビジョニングでの AWS IoT Greengrass Core ソフトウェアのインストールに対するサポートが追加されました。インストール中に、デバイスは AWS IoT に接続して必要な AWS リソースをプロビジョニングし、通常の操作に使用するデバイス証明書をダウンロードします。

リリース: 2021 年 6 月 29 日 AWS IoT Greengrass Core v2.3.0 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.3.0 が提供されています。

リリース日: 2021 年 6 月 29 日

リリースハイライト

- 大規模な設定のサポート - Greengrass nucleus コンポーネントは、最大 10 MB までのデプロイドキュメントをサポートするようになりました。Greengrass コンポーネントに、より大きな設定更新をデプロイできるようになりました。

> **Note**
>
> この機能を使用するには、コアデバイスの AWS IoT ポリシーで、`greengrass:GetDeploymentConfiguration` 権限が許可されなければなりません。リソースをプロビジョニングするための AWS IoT Greengrass Core ソフトウェアインストーラを使用する場合は、コアデバイスの AWS IoT ポリシーで `greengrass:*` が許可されます。これには、この権限が含まれます。詳細については、「AWS IoT Greengrass のデバイス認証と認可」を参照してください。

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.3.0 を利用できます。

コンポーネント	詳細
	新機能 • 7 KB (モノをターゲットとするデプロイの場合) または 31 KB (モノグループをターゲットとするデプロイの場合) から 10 MB までのデプロイ設定ドキュメントのサポートが追加されました。 この機能を使用するには、コアデバイスの AWS IoT ポリシーで、`greengrass:GetDeploymentConfiguration` 権限が許可されなければなりません。リソースをプロビジョニングするための AWS IoT Greengrass Core ソフトウェアインストーラを使用する場合は、コアデバイスの AWS IoT ポリシーで `greengrass:*` が許可されます。これには、この権限が含まれます。詳細については、「AWS IoT Greengrass のデバイス認証と認可」を参照してください。 • `iot:thingName` の recipe 変数が追加されました。この recipe 変数を使用して、recipe 内のコアデバイスの AWS IoT モノの名前を取得できます。詳細については、「レシピ変数」を参照してください。 バグ修正と機能向上 • 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。

リリース: 2021 年 6 月 18 日 AWS IoT Greengrass Core v2.2.0 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.2.0 と AWS から新たに提供されるコンポーネントおよび AWS から提供されるコンポーネントの更新を提供します。

リリース日: 2021 年 6 月 18 日

リリースハイライト

- クライアントデバイスのサポート - AWS から新たに提供されているクライアントデバイスコンポーネントを使用すると、クラウド検出を使用してクライアントデバイスをコアデバイスに接続できます。クライアントデバイスを AWS IoT Core と同期し、Greengrass コンポーネントでクライアントデバイスとやり取りすることができます。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

- ローカルシャドウサービス - 新しいシャドウマネージャーコンポーネントを使用すると、コアデバイスでローカルシャドウサービスを有効にできます。このシャドウサービスを使用して、オフライン時に AWS IoT Device SDK の Greengrass プロセス間通信 (IPC) ライブラリを使用して、ローカルシャドウとやり取りできます。シャドウマネージャーコンポーネントを使用して、ローカルシャドウ状態を AWS IoT Core と同期できるようになります。詳細については、「デバイスシャドウとやり取り」を参照してください。

リリース詳細

- パブリックコンポーネントの更新

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.2.0 を利用できます。 新機能 • ローカルシャドウ管理用の IPC 操作が追加されました。

コンポーネント	詳細
	バグ修正と機能向上 • JAR ファイルのサイズを小さくします。 • メモリ使用量を削減します。 • 特定のケースでログ設定が更新されなかった問題を修正しました。 • 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。
シャドウマネージャー	シャドウマネージャーコンポーネントの新バージョン v 2.0.0 を利用できます。 新機能 • クラシックシャドウと名前付きシャドウのサポートが追加されました。 • IPC を使用したローカルシャドウ管理のサポートが追加されました。 • AWS IoT Core とのシャドウ同期のサポートが追加されました。
クライアントデバイス認証	クライアントデバイス認証コンポーネントの新バージョン v 2.0.0 を利用できます。 新機能 • MQTT 経由でコアデバイスに接続するローカル IoT デバイスである Greengrass クライアントデバイスのサポートが追加されました。 • クライアントデバイスとその MQTT アクションの認証と認可のサポートが追加されました。
モケット MQTT ブローカー	Moquette MQTT ブローカーコンポーネントの新バージョン v 2.0.0 を利用できます。 新機能 • クライアントデバイスとの通信を処理するローカル Moquette MQTT ブローカーのサポートが追加されました。

コンポーネント	詳細
MQTT ブリッジ	MQTT ブリッジコンポーネントの新バージョン v 2.0.0 を利用できます。 新機能 • ローカル MQTT ブローカー、ローカル Greengrass パブリッシュ/サブスクライブブローカー、および AWS IoT Core MQTT ブローカー間でのメッセージのリレーに対するサポートが追加されました。
IP ディテクター	IP ディテクターコンポーネントの新バージョン v 2.0.0 を利用できます。 新機能 • コアデバイスのローカル MQTT ブローカーエンドポイントを、クライアントデバイスが接続するための AWS IoT Greengrass クラウドサービスに報告するためのサポートが追加されました。
ログマネージャー	ログマネージャーコンポーネントのバージョン 2.1.1 を利用できます。 バグ修正と機能向上 • 特定のケースでシステムログ設定が更新されなかった問題を修正します。
DLR オブジェクトの検出	DLR オブジェクトの検出のバージョン 2.1.2 を利用できます。 バグ修正と機能向上 • サンプルの DLR オブジェクト検出推論結果で不正確なバウンディングボックスが発生するイメージスケーリング上の問題を修正しました。
TensorFlow ライトオブジェクト検出	TensorFlow Lite オブジェクト検出のバージョン 2.1.1 を利用できます。 バグ修正と機能向上 • サンプル TensorFlow Lite オブジェクト検出推論結果で不正確な境界ボックスが発生するイメージスケーリングの問題を修正しました。

リリース: 2021 年 4 月 26 日 AWS IoT Greengrass Core v2.1.0 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.1.0 と AWS から提供されるコンポーネントの更新を提供します。

リリース日: 2021 年 4 月 26 日

リリースハイライト

- Docker Hub と Amazon Elastic Container Registry (Amazon ECR) の統合 - 新しい Docker アプリケーションマネージャーコンポーネントを使用すると、Amazon ECR からパブリックイメージまたはプライベートイメージをダウンロードできます。このコンポーネントを使用して、Docker Hub と AWS Marketplace からパブリックイメージをダウンロードすることもできます。詳細については、「Docker コンテナの実行」を参照してください。
- AWS IoT Greengrass Core ソフトウェアの Dockerfile と Docker イメージ - Greengrass Docker イメージを使用して Amazon Linux 2 を基本オペレーティングシステムとして使用する Docker コンテナ内の AWS IoT Greengrass を実行することができます。また、AWS IoT Greengrass Dockerfile を使用して、独自の Greengrass イメージを構築することもできます。詳細については、「Docker コンテナで AWS IoT Greengrass Core ソフトウェアを実行する」を参照してください。
- 追加の機械学習フレームワークとプラットフォームのサポート - TensorFlow Lite 2.5.0 と DLR 1.6.0 を使用してサンプルイメージ分類とオブジェクト検出を実行するために、事前トレーニング済みモデルを使用するサンプル機械学習推論コンポーネントをデプロイできます。このリリースでは、Armv8 (Aarch64) デバイスのサンプル機械学習サポートも拡張されています。詳細については、「機械学習の推論を実行する」を参照してください。

リリース詳細

- プラットフォームサポートの更新
- パブリックコンポーネントの更新

プラットフォームサポートの更新

プラットフォーム	詳細
Docker	AWS IoT Greengrass の Dockerfile と Docker イメージを利用できるようになりました。 Dockerfile AWS IoT Greengrass は、Amazon Linux 2 (x86_64) のベースイメージに AWS IoT Greengrass Core ソフトウェアと依存関係をインストールしたコンテナイメージを構築するための Dockerfile を提供します。異なるプラットフォームアーキテクチャで、Dockerfile のベースイメージを変更して AWS IoT Greengrass を実行することができます。 Docker イメージ AWS IoT Greengrass は Amazon Linux 2 (x86_64) のベースイメージに AWS IoT Greengrass Core ソフトウェアと依存関係をインストールしたビルド済みの Docker イメージ を提供します。 詳細については、「Docker コンテナで AWS IoT Greengrass Core ソフトウェアを実行する」を参照してください。

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。

デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細

Greengrass nucleus	Greengrass nucleus のバージョン 2.1.0 を利用できます。
	
	新機能
	• Amazon ECR のプライベートリポジトリからの Docker イメージのダウンロードがサポートされます。
	• 次のパラメータを追加して、コアデバイスの MQTT 設定をカスタマイズします。
	  • `maxInFlightPublishes` - 同時に送信できる未確認 MQTT QoS 1 メッセージの最大数。
	  • `maxPublishRetry` - パブリッシュに失敗したメッセージを再試行する最大回数。
	• `fleetstatusservice` 設定パラメータを追加して、コアデバイスがデバイスステータスを AWS クラウド に公開する間隔を設定します。
	• 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。
	
	バグ修正と機能向上
	• nucleus が再起動したときにシャドウデプロイが複製される問題を修正しました。
	• サービスロード例外が発生したときに nucleus がクラッシュする問題を修正しました。
	• 循環依存関係を含むデプロイが失敗するように、コンポーネントの依存関係の解決が改善されました。
	• そのコンポーネントがコアデバイスから削除された場合に、プラグインコンポーネントが再デプロイされない問題を修正しました。
	• Lambda コンポーネントや root で実行するコンポーネントの `/greengrass/v2/work` ディレクトリに `HOME` 環境変数が設定される

コンポーネント	詳細
	問題を修正しました。コンポーネントを実行するユーザーのホームディレクトリに HOME 変数が正しく設定されるようになりました。 • 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。
Docker アプリケーションマネージャー	Docker アプリケーションマネージャーコンポーネントの新しいバージョン 2.0.0 を利用できます。 新機能 • Amazon ECR のプライベートリポジトリからイメージをダウンロードするための認証情報を管理します。 • Amazon ECR、Docker Hub、および AWS Marketplace からパブリックイメージをダウンロードします。
Lambda ランチャー	Lambda ランチャーコンポーネントのバージョン 2.0.4 を利用できます。 バグ修正と機能向上 • コンポーネントが `AddGroupOwner` Lambda 関数コンテナを正しく渡さない問題を修正しました。
レガシーサブスクリプションルーター	レガシーサブスクリプションルーターコンポーネントのバージョン 2.1.0 を利用できます。 バグ修正と機能向上 • `source` と `target` の ARN の代わりにコンポーネント名を指定するサポートを追加しました。サブスクリプションのコンポーネント名を指定する場合、Lambda 関数のバージョンが変更されるたびにサブスクリプションを再設定する必要はありません。

コンポーネント	詳細
ローカルデバッグコンソール	ローカルデバッグコンソールコンポーネントのバージョン 2.1.0 を利用できます。 新機能 • HTTPS を使用して、ローカルデバッグコンソールへの接続を保護します。HTTPS はデフォルトで有効になっています。 バグ修正と機能向上 • 設定エディタのフラッシュバーメッセージを消去できます。
ログマネージャー	ログマネージャーコンポーネントのバージョン 2.1.0 を利用できます。 バグ修正と機能向上 • 標準出力 (stdout) と標準エラー (stderr) で出力する Greengrass コンポーネントに機能する `logFileDirectoryPath` と `logFileRegex` にデフォルトを使用します。 • ログを CloudWatch Logs にアップロードするときに、設定済みのネットワークプロキシを介してトラフィックを正しくルーティングします。 • ログストリーム名でコロン文字 (:) を正しく処理します。 CloudWatch Logs ログストリーム名はコロンをサポートしていません。 • ログストリームからモノグループ名を削除して、ログストリーム名を簡素化します。 • 通常の動作中に出力されるエラーログメッセージを削除します。

コンポーネント	詳細
DLR イメージ分類	DLR イメージ分類のバージョン 2.1.1 コンポーネントを利用できます。 新機能 • 深層学習ランタイム のバージョン 1.6.0.を使用します。 • Armv8 (AArch64) プラットフォームでのサンプルイメージ分類のサポートが追加されました。これにより、Jetson Nano などの NVIDIA Jetson を実行している Greengrass コアデバイスにまで機械学習サポートが拡張されます。 • サンプル推論でカメラ統合が可能になりました。新しい `UseCamera` 設定パラメータを使用すると、サンプル推論コードが Greengrass コアデバイスのカメラにアクセスできるようになり、キャプチャしたイメージ上でローカルに推論を実行できるようになります。 • AWS クラウド に推論結果をパブリッシュするためのサポートが追加されました。新しい `PublishResultsOnTopic` 設定パラメータを使用して、結果をパブリッシュするトピックを指定します。 • 推論を実行するイメージ用のカスタムディレクトリを指定できる新しい `ImageDirectory` 設定パラメータが追加されました。 バグ修正と機能向上 • 別の推論ファイルではなく、コンポーネントログファイルに推論結果を書き込みます。 • AWS IoT Greengrass Core ソフトウェアのログモジュールを使用してコンポーネントの出力を記録します。 • AWS IoT Device SDK を使用してコンポーネント設定を読み込み、設定への変更を適用します。

コンポーネント	詳細
DLR オブジェクトの検出	DLR オブジェクトの検出コンポーネントのバージョン 2.1.1 を利用できます。 新機能 • 深層学習ランタイム のバージョン 1.6.0.を使用します。 • Armv8 (AArch64) プラットフォーム上でのサンプルオブジェクト検出のサポートが追加されました。これにより、Jetson Nano などの NVIDIA Jetson を実行している Greengrass コアデバイスにまで機械学習サポートが拡張されます。 • サンプル推論でカメラ統合が可能になりました。新しい `UseCamera` 設定パラメータを使用すると、サンプル推論コードが Greengrass コアデバイスのカメラにアクセスできるようになり、キャプチャしたイメージ上でローカルに推論を実行できるようになります。 • AWS クラウド に推論結果をパブリッシュするためのサポートが追加されました。新しい `PublishResultsOnTopic` 設定パラメータを使用して、結果をパブリッシュするトピックを指定します。 • 推論を実行するイメージ用のカスタムディレクトリを指定できる新しい `ImageDirectory` 設定パラメータが追加されました。 バグ修正と機能向上 • 別の推論ファイルではなく、コンポーネントログファイルに推論結果を書き込みます。 • AWS IoT Greengrass Core ソフトウェアのログモジュールを使用してコンポーネントの出力を記録します。 • AWS IoT Device SDK を使用してコンポーネント設定を読み込み、設定への変更を適用します。

コンポーネント	詳細
DLR イメージ分類モデルストア	DLR イメージ分類モデルストアコンポーネントのバージョン 2.1.1 を利用できます。 新機能 • Armv8 (AArch64) プラットフォーム用のサンプル ResNet-50 イメージ分類モデルを追加します。これにより、Jetson Nano などの NVIDIA Jetson を実行している Greengrass コアデバイスにまで機械学習サポートが拡張されます。
DLR オブジェクト検出モデルストア	DLR オブジェクト検出モデルストアコンポーネントのバージョン 2.1.1 を利用できます。 新機能 • Armv8 (aArch64) プラットフォーム用のサンプル YoLov3 オブジェクト検出モデルが追加されました。これにより、Jetson Nano などの NVIDIA Jetson を実行している Greengrass コアデバイスにまで機械学習サポートが拡張されます。
DLR インストーラ	DLR コンポーネントのバージョン 1.6.1 を利用できます。 新機能 • 深層学習ランタイム v1.6.0 とその依存関係をインストールします。 • Armv8 (AArch64) プラットフォームへの DLR のインストールに対するサポートが追加されました。これにより、Jetson Nano などの NVIDIA Jetson を実行している Greengrass コアデバイスにまで機械学習サポートが拡張されます。 バグ修正と機能向上 • 仮想環境で AWS IoT Device SDK をインストールしてコンポーネント設定を読み取り、設定の変更を適用します。 • マイナーなバグの追加修正と機能向上。

コンポーネント	詳細
TensorFlow ライトイメージ分類	新しい TensorFlow Lite イメージ分類コンポーネントのバージョン 2.1.0 を利用できます。 新機能 • TensorFlow Lite を使用したサンプルイメージ分類推論のサポートを追加します。
TensorFlow ライトオブジェクト検出	新しい TensorFlow Lite オブジェクト検出コンポーネントのバージョン 2.1.0 が利用可能になりました。 新機能 • TensorFlow Lite を使用したサンプルオブジェクト検出推論のサポートを追加します。
TensorFlow Lite イメージ分類モデルストア	新しい TensorFlow Lite イメージ分類モデルストアコンポーネントのバージョン 2.1.0 が利用可能になりました。 新機能 • TensorFlow Lite を使用して、サンプル画像分類推論用の事前トレーニング済み MobileNet v1 量子化モデルを提供します。
TensorFlow Lite オブジェクト検出モデルストア	新しい TensorFlow Lite オブジェクト検出モデルストアコンポーネントのバージョン 2.1.0 が利用可能になりました。 新機能 • TensorFlow Lite を使用したサンプルオブジェクト検出推論のために、COCO データセットでトレーニングされた事前トレーニング済みのシングルショット検出 (SSD) MobileNet モデルを提供します。
TensorFlow ライト	新しい TensorFlow Lite コンポーネントのバージョン 2.5.0 が利用可能になりました。 新機能 • TensorFlow Lite v1.6.0 とその依存関係を Armv7, Armv8および x86_64 プラットフォームの仮想環境にインストールします。AArch64

リリース: 2021 年 3 月 9 日 AWS IoT Greengrass Core v2.0.5 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.0.5 と AWS から提供されるコンポーネントの更新を提供します。ネットワークプロキシのサポートに関する問題と、AWS 中国リージョンの Greengrass データプレーンのエンドポイントに関する問題が修正されています。

リリース日: 2021 年 3 月 9 日

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.0.5 を利用できます。 バグ修正と機能向上 • AWS が提供するコンポーネントのダウンロード時に、設定されたネットワークプロキシを介してトラフィックを正しくルーティングされます。

コンポーネント	詳細
	• AWS 中国リージョンで正しい Greengrass データプレーンのエンドポイントが使用されます。

リリース: 2021 年 2 月 4 日 AWS IoT Greengrass Core v2.0.4 ソフトウェア更新

このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.0.4 が提供されています。これには、ポート 443 経由で HTTPS 通信を設定するための新しい `greengrassDataPlanePort` パラメータとバグの修正が含まれています。`--provision true` で AWS IoT Greengrass Core ソフトウェアインストーラが実行されているときに、最低限の IAM ポリシーに、`iam:GetPolicy` と `sts:GetCallerIdentity` が必要になりました。

リリース日: 2021 年 2 月 4 日

パブリックコンポーネントの更新

次のテーブルは、新機能および更新された機能を含めた AWS の提供するコンポーネントを一覧化したものです。

> **Important**
>
> コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
> デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	詳細
Greengrass nucleus	Greengrass nucleus のバージョン 2.0.4 を利用できます。 新機能 • ポート 443 経由の HTTPS トラフィックが有効になりました。nucleus コンポーネントのバージョン 2.0.4 新しい `greengrassDataPlanePort` 設定パラメータで、デフォルトのポート 8443 ではなくポート 443 を経由するように HTTPS 通信を設定できるようになりました。詳細については、「ポート 443 経由で HTTPS を設定する」を参照してください。 • 作業パスの recipe 変数が追加されました。この recipe 変数を使用して、コンポーネントの作業フォルダへのパスを取得できます。このパスを使用して、コンポーネントとその依存関係間でファイルを共有できます。詳細については、「作業パスの recipe 変数」を参照してください。 バグ修正と機能向上 • ロールポリシーが既に存在する場合に、トークン交換 AWS Identity and Access Management (IAM) ロールポリシーの作成が防止されるようになりました。 この変更の結果、`--provision true` で実行する際にインストーラで `iam:GetPolicy` と `sts:GetCallerIdentity` が必要になります。詳細については、「インストーラがリソースをプロビジョニングするための最小限の IAM ポリシー」を参照してください。 • まだ正常に登録されていないデプロイのキャンセルが正しく処理されるようおになりました。 • デプロイをロールバックするときに、新しいタイムスタンプのある古いエントリが削除されるように設定を更新しました。 • 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。

AWS IoT Greengrass バージョン 1 から移行

AWS IoT Greengrass Version 2 は、AWS IoT Greengrass Core ソフトウェア、API、コンソールの主なバージョンリリースです。AWS IoT Greengrass V2 は、モジュラアプリケーション、デバイスの大規模フリートへのデプロイ、追加プラットフォームに対するサポートなど、AWS IoT Greengrass V1 に対していくつかの改善点を導入しています。

> **Note**
>
> 2023 年 6 月 30 日以降、AWS IoT Greengrass Version 1 では、機能の更新、拡張、バグ修正、またはセキュリティパッチは提供されなくなります。詳細については、「AWS IoT Greengrass V1 メンテナンスポリシー」を参照してください。AWS IoT Greengrass V1 を使用する場合、AWS IoT Greengrass V2 に移行することを強くお勧めします。

このガイドの指示に従って、AWS IoT Greengrass V1 から AWS IoT Greengrass V2 に移行します。

V2 に V1 アプリケーションを実行できますか?

ほとんどの V1 アプリケーションは、アプリケーションコードを変更することなく V2 コアデバイスで実行できます。V1 アプリケーションが次の機能を使用している場合、V2 ではそのアプリケーションを実行できなくなります。

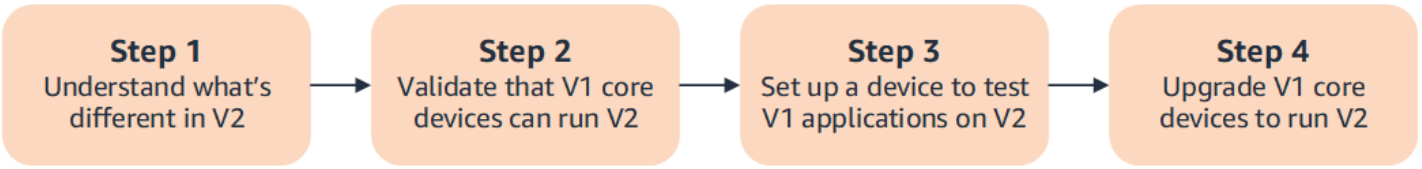
- C と C++ Lambda 関数のランタイム

V1 アプリケーションが次の機能のいずれかを使用する場合、AWS IoT Device SDK V2 を使用して AWS IoT Greengrass V2 にアプリケーションを実行するため、アプリケーションコードを修正する必要があります。

- ローカルシャドウサービスとやり取り
- ローカル接続デバイス (Greengrass デバイス) にメッセージのパブリッシュ

移行の概要

高度なレベルでは、次の手順を使用してコアデバイスを AWS IoT Greengrass V1 から AWS IoT Greengrass V2 にアップグレードできます。実行する正確な手順は、環境の特定要件によって異なります。



1. V1 と V2 の違いを理解

   AWS IoT Greengrass V2 は、デバイスフリートとデプロイ可能なソフトウェアに関する新しい基本概念を導入し、さらに V2 は V1 のいくつかの概念を簡素化します。

   AWS IoT Greengrass V2 クラウドサービスと AWS IoT Greengrass Core ソフトウェア v2.x は、AWS IoT Greengrass V1 クラウドサービスと AWS IoT Greengrass Core ソフトウェア v1.x に対して後方互換性はありません。その結果、 AWS IoT Greengrass V1 over-the-air (OTA) 更新ではコアデバイスを V1 から V2 にアップグレードできません。

2. V1 コアデバイスが V2 を実行できることを検証

   V1 コアデバイスが、AWS IoT Greengrass Core ソフトウェア v2.x と AWS IoT Greengrass V2 機能を実行できることを検証します。AWS IoT Greengrass V2 は AWS IoT Greengrass V1 とは異なるデバイス要件があります。

3. V2 で V1 アプリケーションをテストする新しいデバイスをセットアップ

   本番環境のデバイスのリスクを最小限に抑えるには、V2 で V1 アプリケーションをテストする新しいデバイスを作成します。AWS IoT Greengrass Core ソフトウェア v2.x をインストールした後、AWS IoT Greengrass V2 コンポーネントを作成して展開して、AWS IoT Greengrass V1 アプリケーションを移行してテストできます。

4. V1 コアデバイスをアップグレードして V2 を実行

   既存の V1 コアデバイスをアップグレードして、AWS IoT Greengrass Core ソフトウェア v2.x と AWS IoT Greengrass V2 コンポーネントを実行します。V1 から V2 にデバイスのフリートを移行するには、フリートの各デバイスに対してこの手順を繰り返します。

AWS IoT Greengrass V1 と の違い AWS IoT Greengrass V2

AWS IoT Greengrass V2 では、デバイス、フリート、デプロイ可能なソフトウェアに関する新しい基本概念が導入されました。このセクションでは、V2 では異なる V1 の概念について説明します。

Greengrass の概念と用語

概念	AWS IoT Greengrass V1	AWS IoT Greengrass V2
アプリケーションコード	では AWS IoT Greengrass V1、Lambda 関数はコアデバイスで実行されるソフトウェアを定義します。各 Greengrass グループで、関数が使用するサブスクリプションとローカルリソースを定義します。 AWS IoT Greengrass Core ソフトウェアがコンテナ化された Lambda ランタイム環境で実行される Lambda 関数の場合、メモリ制限などのコンテナパラメータを定義します。	では AWS IoT Greengrass V2、コンポーネントはコアデバイスで実行されるソフトウェアモジュールです。 • 各コンポーネントにはコンポーネントのライフサイクルの各段階で実行するメタデータや、パラメータ、依存関係、スクリプトを定義した recipe があります。 • recipe は、コンポーネントのアーティファクトも定義します。これらは、スクリプト、コンパイルされたコード、静的リソースなどのバイナリファイルです。 • コアデバイスにコンポーネントをデプロイすると、コアデバイスによってコンポーネント recipe とアーティファクトがダウンロードされ、コンポーネントが実行されます。 AWS IoT Greengrass V2 で Lambda ランタイム環境で実行されるコンポーネントとし

概念	AWS IoT Greengrass V1	AWS IoT Greengrass V2
		て V1 Lambda 関数をインポートできます。Lambda 関数をインポートするときは、関数のサブスクリプション、ローカルリソース、およびコンテナパラメータを指定します。詳細については、「ステップ 2: AWS IoT Greengrass V2 コンポーネントを作成してデプロイし、 AWS IoT Greengrass V1 アプリケーションを移行する」を参照してください。 カスタムコンポーネントを作成する方法の詳細については、「AWS IoT Greengrass コンポーネントを開発する」を参照してください。

概念	AWS IoT Greengrass V1	AWS IoT Greengrass V2
AWS IoT Greengrass グループとデプロイ	では AWS IoT Greengrass V1、グループはコアデバイス、そのコアデバイスの設定とソフトウェア、およびそのコアデバイスに接続できる AWS IoT モノのリストを定義します。グループの設定をコアデバイスに送信するデプロイを作成します。	では AWS IoT Greengrass V2、デプロイを使用して、コアデバイスで実行されるソフトウェアコンポーネントと設定を定義します。 • 各デプロイは、単一のコアデバイス (AWS IoT モノ) または複数のコアデバイスを含むことができる AWS IoT モノのグループを対象としています。 • モノグループへのデプロイは継続的であるため、コアデバイスをモノグループに追加すると、そのグループのソフトウェア設定を受信します。 詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。 では AWS IoT Greengrass V2、Greengrass CLI を使用してローカルデプロイを作成し、開発するデバイスでカスタムソフトウェアコンポーネントをテストすることもできます。詳細については、「AWS IoT Greengrass コンポーネントの作成」を参照してください。

概念	AWS IoT Greengrass V1	AWS IoT Greengrass V2
AWS IoT Greengrass Core ソフトウェア	では AWS IoT Greengrass V1、 AWS IoT Greengrass Core ソフトウェアは、ソフトウェアとそのすべての機能を含む単一のパッケージです。 AWS IoT Greengrass Core ソフトウェアをインストールするエッジデバイスは Greengrass コアと呼ばれます。	では AWS IoT Greengrass V2、 AWS IoT Greengrass Core ソフトウェアはモジュール式であるため、メモリフットプリントを制御するために何をインストールするかを選択できます。 • Greengrass nucleus コンポーネントは、 AWS IoT Greengrass Core ソフトウェアのインストールに最低限必要なコンポーネントです。nucleus をインストールするエッジデバイスは、Greengrass コアデバイスと呼ばれます。 • nucleus は、コアデバイス上の他のコンポーネントのデプロイ、オーケストレーション、およびライフサイクル管理を行います。 • ストリームマネージャー、シークレットマネージャー、ログマネージャーなどの機能は、それらの機能が必要な場合にのみデプロイするコンポーネントです。詳細については、「AWS が提供したコンポーネント」を参照してください。

概念	AWS IoT Greengrass V1	AWS IoT Greengrass V2
Connector	では AWS IoT Greengrass V1、コネクタは、ローカルインフラストラクチャ、デバイスプロトコル、およびその他のクラウドサービスとやり取りするために AWS IoT Greengrass V1 コアデバイスにデプロイする構築済みのモジュール AWSです。	では AWS IoT Greengrass V2、 は V1 のコネクタが提供する機能を実装する Greengrass コンポーネント AWS を提供します。以下の AWS IoT Greengrass V2 コンポーネントは Greengrass V1 コネクタ機能を提供します。 • CloudWatch メトリクスコンポーネント • AWS IoT Device Defender コンポーネント • Firehose コンポーネント • Modbus-RTU プロトコルアダプタコンポーネント • Amazon SNS コンポーネント 詳細については、「AWS が提供したコンポーネント」を参照してください。

概念	AWS IoT Greengrass V1	AWS IoT Greengrass V2
接続デバイス (Greengrass デバイス)	では AWS IoT Greengrass V1、接続されたデバイスとは、Greengrass グループに追加して、そのグループ内のコアデバイスに接続し、MQTT 経由で通信する AWS IoT モノです。接続デバイスを追加または削除するたびに、そのグループをデプロイする必要があります。サブスクリプションを使用して、接続されたデバイス AWS IoT Coreとコアデバイスのアプリケーション間でメッセージをリレーします。	では AWS IoT Greengrass V2、接続されたデバイスは Greengrass クライアントデバイスと呼ばれます。 • クライアントデバイスをコアデバイスに関連付けて、それらを接続し、MQTT 経由で通信します。 • クライアントデバイスの接続を認可するには、クライアントデバイスのグループに適用できる認証ポリシーを定義します。そのため、クライアントデバイスを追加または削除するためのデプロイを作成する必要はありません。 • クライアントデバイス AWS IoT Coreと Greengrass コンポーネント間でメッセージをリレーするには、オプションの MQTT ブリッジコンポーネントを設定できます。 AWS IoT Greengrass V1 と の両方で AWS IoT Greengrass V2、デバイスは FreeRTOS を実行するか、 AWS IoT Device SDK または Greengrass 検出 API を使用して、接続できるコアデバイスに関する情報を取得できます。Greengrass

概念	AWS IoT Greengrass V1	AWS IoT Greengrass V2
		Discovery API は下位互換性があるため、V1 コアデバイスに接続するクライアントデバイスがある場合は、コードを変更せずに V2 コアデバイスに接続できます。 デバイスクラスの詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。
ローカルリソース	では AWS IoT Greengrass V1、コンテナで実行される Lambda 関数は、コアデバイスのファイルシステム上のボリュームとデバイスにアクセスするように設定できます。これらのファイルシステムリソースは、ローカルリソースと呼ばれます。	では AWS IoT Greengrass V2、Lambda 関数 、Docker コンテナ 、ネイティブオペレーティングシステムプロセス またはカスタムランタイム であるコンポーネントを実行できます。 • コンテナ化された Lambda 関数をコンポーネントとしてインポートする場合は、関数が使用するローカルリソースを指定する必要があります。 • コンテナ化されていない Lambda 関数と Lambda 以外のコンポーネントは、コアデバイス上のローカルリソースを直接操作できるため、コンポーネントが使用するローカルリソースを指定する必要はありません。

概念	AWS IoT Greengrass V1	AWS IoT Greengrass V2
ローカルシャドウサービス	では AWS IoT Greengrass V1、ローカルシャドウサービスはデフォルトで有効になっており、名前のないクラシックシャドウのみをサポートします。Lambda 関数で AWS IoT Greengrass Core SDK を使用して、デバイスのシャドウを操作します。	では AWS IoT Greengrass V2、シャドウマネージャーコンポーネントをデプロイしてローカルシャドウサービスを有効にします。 • AWS IoT Device SDK V2 を Lambda 関数とカスタムコンポーネントで使用して、デバイスのシャドウとやり取りできます。 • ローカルシャドウサービスは、名前付きシャドウをサポートします。 • ローカルシャドウサービスを使用すると、シャドウを削除し、削除されたシャドウを と同期できます AWS IoT Core。 詳細については、「デバイスシャドウとやり取り」を参照してください。

概念	AWS IoT Greengrass V1	AWS IoT Greengrass V2

サブスクリプション	では AWS IoT Greengrass V1、Greengrass グループのサブスクリプションを定義して、Lambda 関数、コネクタ、接続されたデバイス、AWS IoT Core MQTT ブローカー、ローカルシャドウサービス間の通信チャネルを指定します。サブスクリプションは、関数ペイロードとして消費するイベントメッセージを Lambda 関数が受け取る場所を指定します。	では AWS IoT Greengrass V2、サブスクリプションを使用せずに通信チャネルを指定します。
		• コンポーネントは独自の通信チャネルを管理し、ローカルのパブリッシュ/サブスクライブメッセージや、AWS IoT Core MQTT メッセージ、ローカルシャドウサービスとやり取りします。
		  • 別のコンポーネントまたは AWS IoT Core MQTT ブローカーからのメッセージに反応するコンポーネントを開発するには、ローカルのパブリッシュ/サブスクライブメッセージングと AWS IoT Core MQTT メッセージングにプロセス間通信 (IPC) インターフェイスを使用できます。
		  • ローカルシャドウサービスと対話するコンポーネントを開発するには、ローカルシャドウサービスの IPC インターフェイスを使用できます。
		  • コンポーネントコンフィギュレーションで、承認

概念	AWS IoT Greengrass V1	AWS IoT Greengrass V2
		ポリシーを定義して、コンポーネントが使用する権限を持つトピックとローカルシャドウを指定します。 • クライアントデバイス、ローカルパブリッシュ/サブスクライブブローカー、および AWS IoT Core MQTT ブローカー間の通信チャネルを設定するには、MQTT ブリッジコンポーネントを設定してデプロイします。MQTT ブリッジコンポーネントを使用すると、コンポーネント内のクライアントデバイスとやり取りし、クライアントデバイスと AWS IoT Core の間でメッセージを中継できます。

概念	AWS IoT Greengrass V1	AWS IoT Greengrass V2
他の へのアクセス AWS のサービス	では AWS IoT Greengrass V1、グループロールと呼ばれる AWS Identity and Access Management (IAM) ロールを Greengrass グループにアタッチします。グループロールは、そのグループのコアデバイスの Lambda 関数と AWS IoT Greengrass 機能が にアクセスするために使用するアクセス許可を定義します AWS のサービス。	では AWS IoT Greengrass V2、 AWS IoT ロールエイリアスを Greengrass コアデバイスにアタッチします。ロールエイリアスは、トークン交換ロールと呼ばれる IAM ロールを参照します。トークン交換ロールは、コアデバイス上の Greengrass コンポーネントが AWS のサービス にアクセスするために使用する許可を定義します。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

V1 コアデバイスが V2 ソフトウェアを実行できることを検証

AWS IoT Greengrass Core ソフトウェア v2.x は、AWS IoT Greengrass Core ソフトウェア v1.x とは異なる要件を持っています。V1 コアデバイスを V2 にアップグレードする前に、AWS IoT Greengrass V2 のデバイス要件を確認してください。AWS IoT Greengrass V2 は現在、Yocto プロジェクトを使用してカスタム Linux ベースのシステムの移行をサポートしていません。

AWS IoT Greengrass V2 用 AWS IoT Device Tester (IDT)を使用して、デバイスが AWS IoT Greengrass Core ソフトウェア v2.x を実行する要件を満たしていることを検証します。IDT は、ホストコンピュータに実行されて、検証対象のデバイスに接続するダウンロード可能なテストフレームワークです。IDT を使用して AWS IoT Greengrass 資格スイートを実行するため、指示に従います。IDT を設定するとき、Docker、機械学習 (ML)、データストリーム管理、ハードウェアセキュリティ統合など、デバイスがオプション機能をサポートしているかどうか検証する選択ができます。

IDT が V1 コアデバイスの V2 テストの失敗またはエラーを報告する場合、そのデバイスを V1 から V2 にアップグレードすることはできません。

V1 アプリケーションをテストする新しい V2 コアデバイスをセットアップする

AWS IoT Greengrass V1 アプリケーション用に AWSが提供するコンポーネントと機能をデプロイして AWS Lambda テストするために、新しい AWS IoT Greengrass V2 コアデバイスをセットアップします。この V2 コアデバイスを使用して、コアデバイスでネイティブプロセスを実行する追加のカスタム Greengrass コンポーネントを開発およびテストすることもできます。V2 コアデバイスでアプリケーションをテストした後、既存の V1 コアデバイスを V2 にアップグレードし、V1 機能を提供する V2 コンポーネントをデプロイできます。

ステップ 1: 新しいデバイスに AWS IoT Greengrass V2 をインストールする

Core AWS IoT Greengrass ソフトウェア v2.x を新しいデバイスにインストールします。[getting started tutorial] (入門チュートリアル) に従ってデバイスをセットアップし、コンポーネントを開発およびデプロイする方法を学ぶことができます。このチュートリアルでは、[automatic provisioning] (自動プロビジョニング) を使用して、デバイスをすばやくセットアップできます。 AWS IoT Greengrass Core ソフトウェア v2.x をインストールするときは、Greengrass CLI をデプロイする `--deploy-dev-tools`引数を指定して、コンポーネントをデバイス上で直接開発、テスト、デバッグできるようにします。プロキシの背後で AWS IoT Greengrass Core ソフトウェアをインストールする方法やハードウェアセキュリティモジュール (HSM) を使用する方法など、その他のインストールオプションの詳細については、「」を参照してくださいAWS IoT Greengrass Core ソフトウェアをインストールします。。

(オプション) Amazon CloudWatch Logs へのログ記録を有効にする

V2 コアデバイスが Amazon CloudWatch Logs にログをアップロードできるようにするには、 AWS が提供するログマネージャーコンポーネント をデプロイできます。 CloudWatch Logs を使用してコンポーネントログを表示できるため、コアデバイスのファイルシステムにアクセスせずにデバッグとトラブルシューティングを行うことができます。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

ステップ 2: AWS IoT Greengrass V2 コンポーネントを作成してデプロイし、 AWS IoT Greengrass V1 アプリケーションを移行する

ほとんどの AWS IoT Greengrass V1 アプリケーションは で実行できます AWS IoT Greengrass V2。Lambda 関数は、 で実行されるコンポーネントとしてインポートでき AWS IoT Greengrass

V2、 AWS IoT Greengrass コネクタと同じ機能を提供する AWSが提供するコンポーネントを使用できます。

カスタムコンポーネントを開発して、Greengrass コアデバイスで実行する任意の機能またはランタイムをビルドすることもできます。コンポーネントをローカルで開発およびテストする方法の詳細については、「AWS IoT Greengrass コンポーネントの作成」を参照してください。

トピック

- V1 Lambda 関数をインポートする
- V1 コネクタを使用する
- Docker コンテナを実行する
- 機械学習推論を実行する
- V1 Greengrass デバイスを接続する
- ローカルシャドウサービスを有効にする
- との統合 AWS IoT SiteWise

V1 Lambda 関数をインポートする

Lambda 関数を AWS IoT Greengrass V2 コンポーネントとしてインポートできます。以下のアプローチのいずれかを選択できます。

- V1 Lambda 関数を Greengrass コンポーネントとして直接インポートします。
- AWS IoT Device SDK v2 の Greengrass ライブラリを使用するように Lambda 関数を更新し、Lambda 関数を Greengrass コンポーネントとしてインポートします。
- Lambda 以外のコードと AWS IoT Device SDK v2 を使用して、Lambda 関数と同じ機能を実装するカスタムコンポーネントを作成します。

Lambda 関数がストリームマネージャーやローカルシークレットなどの機能を使用する場合は、これらの機能をパッケージ化する AWSが提供するコンポーネントへの依存関係を定義する必要があります。Lambda 関数コンポーネントをデプロイすると、依存関係として定義する各機能のコンポーネントもデプロイに含まれます。デプロイでは、コアデバイスにデプロイするシークレットなどのパラメータを設定できます。V1 のすべての機能で、V2 の Lambda 関数にコンポーネントの依存関係が必要なわけではありません。次のリストは、V2 Lambda 関数コンポーネントで V1 機能を使用する方法を説明しています。

- 他の AWS サービスにアクセスする

Lambda 関数が AWS 認証情報を使用して他の AWS サービスにリクエストを行う場合、コアデバイスのトークン交換ロールは、コアデバイスが Lambda 関数が使用する AWS オペレーションを実行することを許可する必要があります。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

- ストリームマネージャー

  Lambda 関数がストリームマネージャーを使用している場合は、関数をインポートするときのコンポーネントの依存関係として `aws.greengrass.StreamManager` を指定します。ストリームマネージャーコンポーネントをデプロイするときに、ターゲットコアデバイスに設定するストリームマネージャーパラメータを指定します。コアデバイスのトークン交換ロールは、コアデバイスがストリームマネージャーで使用する AWS クラウド 送信先にアクセスできるようにする必要があります。詳細については、「ストリームマネージャー」を参照してください。

- ローカルシークレット

  Lambda 関数がローカルシークレットを使用している場合は、関数をインポートするときのコンポーネントの依存関係として `aws.greengrass.SecretManager` を指定します。シークレットマネージャーコンポーネントをデプロイするときは、ターゲットコアデバイスにデプロイするシークレットリソースを指定します。コアデバイスのトークン交換ロールは、コアデバイスがデプロイするシークレットリソースを取得できるようにする必要があります。詳細については、「シークレットマネージャー」を参照してください。

  Lambda 関数コンポーネントをデプロイするときは、 AWS IoT Device SDK V2 で IPC オペレーションを使用するアクセス許可を付与する IPC 承認ポリシーを持つように設定します。GetSecretValue

- ローカルシャドウ

  Lambda 関数がローカルシャドウとやり取りする場合は、 AWS IoT Device SDK V2 を使用するように Lambda 関数コードを更新する必要があります。また、関数をインポートするときのコンポーネントの依存関係として `aws.greengrass.ShadowManager` を指定する必要があります。詳細については、「デバイスシャドウとやり取り」を参照してください。

  Lambda 関数コンポーネントをデプロイするときは、 AWS IoT Device SDK V2 でシャドウ IPC オペレーションを使用するアクセス許可を付与する IPC 承認ポリシーを持つように設定してください。

- サブスクリプション

- Lambda 関数がクラウドソースからのメッセージをサブスクライブする場合は、関数をインポートするときにそれらのサブスクリプションをイベントソースとして指定します。
- Lambda 関数が別の Lambda 関数からのメッセージをサブスクライブしている場合、または Lambda 関数が AWS IoT Core または他の Lambda 関数にメッセージを発行する場合は、Lambda 関数をデプロイするときにレガシーサブスクリプションルーターコンポーネントを設定してデプロイします。レガシーサブスクリプションルーターコンポーネントをデプロイするときは、Lambda 関数が使用するサブスクリプションを指定します。

  > **Note**
  >
  > レガシーサブスクリプションルーターコンポーネントは、Lambda 関数が AWS IoT Greengrass Core SDK の `publish()`関数を使用する場合にのみ必要です。 AWS IoT Device SDK V2 でプロセス間通信 (IPC) インターフェイスを使用するように Lambda 関数コードを更新する場合、レガシーサブスクリプションルーターコンポーネントをデプロイする必要はありません。詳細については、次のプロセス間通信サービスを参照してください。
  >
  > - ローカルメッセージをパブリッシュ/サブスクライブする
  > - AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする

- Lambda 関数がローカルに接続されたデバイスからのメッセージをサブスクライブする場合は、関数をインポートするときにそれらのサブスクリプションをイベントソースとして指定します。また、接続されたデバイスからイベントソースとして指定したローカルのパブリッシュ/サブスクライブトピックにメッセージを中継するために、MQTT ブリッジコンポーネントを設定およびデプロイする必要があります。
- Lambda 関数がローカルに接続されたデバイスにメッセージを発行する場合は、 AWS IoT Device SDK V2 を使用してローカルのパブリッシュ/サブスクライブメッセージを発行するように Lambda 関数コードを更新する必要があります。また、ローカルのパブリッシュ/サブスクライブメッセージブローカーから接続されたデバイスにメッセージを中継するために、[MQTT bridge component] (MQTT ブリッジコンポーネント) を設定およびデプロイする必要があります。

- ローカルボリュームとデバイス

  コンテナ化された Lambda 関数がローカルボリュームまたはデバイスにアクセスする場合は、Lambda 関数をインポートするときにそれらのボリュームとデバイスを指定します。この機能では、コンポーネントの依存関係は必要ありません。

詳細については、「AWS Lambda 関数を実行する」を参照してください。

V1 コネクタを使用する

一部の AWS IoT Greengrass コネクタと同じ機能を提供する AWSが提供するコンポーネントをデプロイできます。デプロイを作成するときに、コネクタのパラメータを設定できます。

以下の AWS IoT Greengrass V2 コンポーネントは Greengrass V1 コネクタ機能を提供します。

- CloudWatch メトリクスコンポーネント
- AWS IoT Device Defender コンポーネント
- Firehose コンポーネント
- Modbus-RTU プロトコルアダプタコンポーネント
- Amazon SNS コンポーネント

Docker コンテナを実行する

AWS IoT Greengrass V2 には、V1 Docker アプリケーションデプロイコネクタを直接置き換えるコンポーネントはありません。ただし、Docker アプリケーションマネージャーコンポーネントを使用して Docker イメージをダウンロードし、ダウンロードしたイメージから Docker コンテナを実行するカスタムコンポーネントを作成できます。詳細については、Docker コンテナの実行およびDocker アプリケーションマネージャーを参照してください。

機械学習推論を実行する

AWS IoT Greengrass V2 は、Amazon SageMaker Edge Manager エージェントをインストールし、SageMaker Neo でコンパイルされたモデルを Greengrass コアデバイスのモデルコンポーネントとして使用できる Amazon SageMaker Edge Manager コンポーネントを提供します。 は、デバイスに Deep Learning Runtime と TensorFlow Lite をインストールするコンポーネント AWS IoT Greengrass V2 も提供します。対応する DLR および TensorFlow Lite モデルおよび推論コンポーネントを使用して、サンプル画像分類およびオブジェクト検出推論を実行できます。MXNet や などの他の機械学習フレームワークを使用するには TensorFlow、これらのフレームワークを使用する独自のカスタムコンポーネントを開発できます。

V1 Greengrass デバイスを接続する

の接続されたデバイスは、 でクライアントデバイス AWS IoT Greengrass V1 と呼ばれます AWS IoT Greengrass V2。クライアントデバイスの AWS IoT Greengrass V2 サポートは と下位互換性が

あるため AWS IoT Greengrass V1、アプリケーションコードを変更せずに V1 クライアントデバイスを V2 コアデバイスに接続できます。クライアントデバイスが V2 コアデバイスに接続できるようにするには、クライアントデバイスのサポートを有効にする Greengrass コンポーネントをデプロイし、クライアントデバイスをコアデバイスに関連付けます。クライアントデバイス、 AWS IoT Core クラウドサービス、Greengrass コンポーネント (Lambda 関数を含む) 間でメッセージを中継するには、[MQTT bridge component] (MQTT ブリッジコンポーネント) をデプロイおよび設定します。[IP detector component] (IP 検出コンポーネント) をデプロイして接続情報を自動検出することも、エンドポイントを手動で管理することも可能です。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

ローカルシャドウサービスを有効にする

では AWS IoT Greengrass V2、ローカルシャドウサービスは、 AWSが提供するシャドウマネージャーコンポーネントによって実装されます。 には、名前付きシャドウのサポート AWS IoT Greengrass V2 も含まれています。コンポーネントがローカルシャドウとやり取りし、シャドウの状態を に同期できるようにするには AWS IoT Core、シャドウマネージャーコンポーネントを設定してデプロイし、コンポーネントコードでシャドウ IPC オペレーションを使用します。詳細については、「デバイスシャドウとやり取り」を参照してください。

との統合 AWS IoT SiteWise

V1 コアデバイスを AWS IoT SiteWise ゲートウェイとして使用している場合は、手順に従って新しい V2 コアデバイスを AWS IoT SiteWise ゲートウェイとしてセットアップします。 AWS IoT SiteWise には、 AWS IoT SiteWise コンポーネントをデプロイするインストールスクリプトが用意されています。

ステップ 3: AWS IoT Greengrass V2 アプリケーションをテストする

V2 コンポーネントを作成して新しい V2 コアデバイスにデプロイしたら、アプリケーションが期待を満たしていることを確認します。デバイスのログをチェックして、コンポーネントの標準出力 (stdout) および標準エラー (stderr) メッセージを表示できます。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

Greengrass CLI をコアデバイスにデプロイした場合は、それを使用してコンポーネントとその設定をデバッグできます。詳細については、「Greengrass CLI コマンド」を参照してください。

アプリケーションが V2 コアデバイスで動作することを確認したら、アプリケーションの Greengrass コンポーネントを他のコアデバイスにデプロイできます。ネイティブプロセスまたは

Docker コンテナを実行するカスタムコンポーネントを開発する場合は、まずそれらのコンポーネントを サービスにパブリッシュして、他のコアデバイスにデプロイする必要があります。 AWS IoT Greengrass

Greengrass V1 コアデバイスを Greengrass V2 にアップグレードします。

アプリケーションとコンポーネントが AWS IoT Greengrass V2 コアデバイスで作動することを確認したら、本番デバイスなど、現在 v1.x を実行しているデバイスに AWS IoT Greengrass Core ソフトウェア v2.x をインストールできます。次に、Greengrass V2 コンポーネントをデプロイして、Greengrass アプリケーションをデバイスで実行します。

デバイスのフリートを V1 から V2 にアップグレードするには、アップグレードするデバイスごとに次の手順を実行します。モノグループを使用して、V2 コンポーネントをコアデバイスのフリートにデプロイできます。

> **Tip**
>
> デバイスのフリートに対するアップグレードプロセスを自動化するスクリプトを作成することをお勧めします。AWS Systems Manager を使用してフリートを管理する場合、システムマネージャーを使用し、各デバイスにそのスクリプトを実行してフリートを V1 から V2 にアップグレードできます。
> AWS エンタープライズサポートの担当に連絡して、アップグレードプロセスを最適に自動化する方法についてお問い合わせいただけます。

ステップ 1。 AWS IoT Greengrass Core ソフトウェア v2.x をインストール

AWS IoT Greengrass Core ソフトウェア v2.x を V1 コアデバイスにインストールする次のオプションから選択します:

- より少ないステップでアップグレード

  より少ない手順でアップグレードするには、v2.x ソフトウェアをインストールする前に v1.x ソフトウェアをアンインストールできます。
- 最小限のダウンタイムでアップグレード

最小限のダウンタイムでアップグレードするには、AWS IoT Greengrass Core ソフトウェアの両バージョンを同時にインストールできます。AWS IoT Greengrass Core ソフトウェア v2.x をインストールして V2 アプリケーションが正しく動作することを確認したら、AWS IoT Greengrass Core ソフトウェア v1.x をアンインストールします。このオプションを選択する前に、AWS IoT Greengrass Core ソフトウェアの両バージョンを同時に実行するために必要な追加 RAM を検討してください。

v2.x をインストールする前に、AWS IoT Greengrass Core v1.x をアンインストール

順次アップグレードする場合、デバイスに v2.x をインストールする前に、AWS IoT Greengrass Core ソフトウェア v1.x をアンインストールします。

AWS IoT Greengrass Core ソフトウェア v1.x をアンインストールするには

1. AWS IoT Greengrass Core ソフトウェア v1.x がサービスとして実行されている場合、サービスを停止、無効化、削除する必要があります。

   a. 実行中の AWS IoT Greengrass Core ソフトウェア v1.x サービスを停止します。

   ```
   sudo systemctl stop greengrass
   ```

   b. サービスが停止するまで待ちます。`list` コマンドを使用してサービスの状況を確認できます。

   ```
   sudo systemctl list-units --type=service | grep greengrass
   ```

   c. サービスを無効にします。

   ```
   sudo systemctl disable greengrass
   ```

   d. サービスを削除します。

   ```
   sudo rm /etc/systemd/system/greengrass.service
   ```

2. AWS IoT Greengrass Core ソフトウェア v1.x がサービスとして実行されていない場合、次のコマンドを実行してデーモンを停止します。*`greengrass-root`* を Greengrass ルートフォルダの名前に置き換えます。デフォルトの場所は `/greengrass` です。

   ```
   cd /greengrass-root/ggc/core/
   ```

```
sudo ./greengrassd stop
```

3. (オプション) Greengrass ルートフォルダをバックアップし、該当する場合、カスタム書き込みフォルダをデバイスの別のフォルダにバックアップします。

   a. 次のコマンドを使用し、現在の Greengrass ルートフォルダを別のフォルダにコピーして、ルートフォルダを削除します。

   ```
   sudo cp -r /greengrass-root /path/to/greengrass-backup
   rm -rf /greengrass-root
   ```

   b. 次のコマンドを使用し、書き込みフォルダを別のフォルダに移動して、書き込みフォルダを削除します。

   ```
   sudo cp -r /write-directory /path/to/write-directory-backup
   rm -rf /write-directory
   ```

次に、AWS IoT Greengrass V2 のインストール手順を使用して、デバイスにソフトウェアをインストールします。

> **Tip**
>
> コアデバイスを V1 から V2 に移行する際にコアデバイスの ID を再利用するには、手動プロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする指示に従ってください。最初にデバイスから V1 コアソフトウェアを削除してから、V1 コアデバイスの AWS IoT モノと証明書を再利用し、証明書の AWS IoT ポリシーを更新して、v2.x ソフトウェアが必要とする許可を付与します。

既に v1.x を実行しているデバイスに AWS IoT Greengrass Core ソフトウェア v2.x をインストール

既に AWS IoT Greengrass Core ソフトウェア v1.x が実行されているデバイスに AWS IoT Greengrass Core v2.x ソフトウェアをインストールする場合、以下の点に注意してください:

- V2 コアデバイスの AWS IoT モノ名は一意である必要があります。V1 コアデバイスと同じモノ名を使用しないでください。

- AWS IoT Greengrass Core ソフトウェア v2.x に使用するポートは、v1.x に使用するポートとは異なる必要があります。
  - 8088 以外のポートを使用するように V1 ストリームマネージャーを設定します。詳細については、「ストリームマネージャーの設定」を参照してください。
  - 8883 以外のポートを使用するように V1 MQTT ブローカを設定します。詳細については、「ローカルメッセージング用に MQTT ポートの設定」を参照してください。
- AWS IoT Greengrass V2は、Greengrass システムサービスの名前を変更するオプションはありません。Greengrass をシステムサービスとして実行する場合、システムサービス名の不一致を避けるため、次のいずれかの操作を行う必要があります。
  - v2.x をインストールする前に、v1.x のGreengrass サービスの名前を変更します。
  - システムサービスがない AWS IoT Greengrass Core ソフトウェア v2.x をインストールし、`greengrass` 以外の名前で手動でソフトウェアをシステムサービスとして設定します。

v1.x の Greengrass サービスの名前を変更するには

1. AWS IoT Greengrass Core ソフトウェア v1.x サービスを停止します。

   ```
   sudo systemctl stop greengrass
   ```

2. サービスが停止するのを待ちます。サービスの停止に数分かかることがあります。`list-units` コマンドを実行して、サービスが停止したかどうか調べられます。

   ```
   sudo systemctl list-units --type=service | grep greengrass
   ```

3. サービスを無効にします。

   ```
   sudo systemctl disable greengrass
   ```

4. サービスの名前を変更します。

   ```
   sudo mv /etc/systemd/system/greengrass.service /etc/systemd/system/greengrass-v1.service
   ```

5. サービスをリロードして起動します。

   ```
   sudo systemctl daemon-reload
   sudo systemctl reset-failed
   sudo systemctl enable greengrass-v1
   ```

```
sudo systemctl start greengrass-v1
```

次に、AWS IoT Greengrass V2 のインストール手順を使用して、デバイスにソフトウェアをインストールします。

> **Tip**
>
> コアデバイスを V1 から V2 に移行する際にコアデバイスの ID を再利用するには、手動プロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする指示に従ってください。最初にデバイスから V1 コアソフトウェアを削除してから、V1 コアデバイスの AWS IoT モノと証明書を再利用し、証明書の AWS IoT ポリシーを更新して、v2.x ソフトウェアが必要とする許可を付与します。

ステップ 2: コアデバイスに AWS IoT Greengrass V2 コンポーネントをデプロイ

デバイスに AWS IoT Greengrass Core ソフトウェア v2.x をインストールしたら、次のリソースを含むデプロイを作成します。コンポーネントを同様なデバイスのフリートにデプロイするには、それらのデバイスを含むモノグループのデプロイを作成します。

- V1 Lambda 関数から作成した Lambda 関数のコンポーネント。詳細については、「AWS Lambda 関数を実行する」を参照してください。
- V1サブスクリプションを使用する場合、レガシー サブスクリプション ルータ コンポーネント。
- ストリームマネージャーを使用する場合、ストリーム マネージャー コンポーネント。詳細については、「Greengrass コアデバイスでのデータストリームの管理」を参照してください。
- ローカルシークレットを使う場合、シークレットマネージャーコンポーネント。
- V1 コネクタを使用する場合、AWS が提供するコネクタコンポーネント。
- Docker コンテナを使用する場合、Docker アプリケーション マネージャー コンポーネント。詳細については、「Docker コンテナの実行」を参照してください。
- 機械学習の推論を使用する場合、機械学習サポート用コンポーネント。詳細については、「機械学習の推論を実行する」を参照してください。
- 接続されたデバイスを使用する場合、クライアント デバイス サポート用コンポーネント。クライアントデバイスのサポートを有効にして、クライアントデバイスをコアデバイスに関連付ける必要があります。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

- デバイスシャドウを使用する場合、シャドウ マネージャー コンポーネント。詳細については、「デバイスシャドウとやり取り」を参照してください。
- Greengrass コアデバイスから Amazon CloudWatch Logs にログをアップロードすると、ログマネージャーコンポーネント になります。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。
- AWS IoT SiteWiseと統合する場合、指示に従って V2 コアデバイスを AWS IoT SiteWise ゲートウェイとしてセットアップします。AWS IoT SiteWise は、ユーザーに代わって AWS IoT SiteWise コンポーネントをデプロイするインストールスクリプトを提供します。
- カスタム機能を実装するために開発したユーザー定義のコンポーネント。

デプロイの作成と改訂の情報については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

チュートリアル: AWS IoT Greengrass V2 の開始方法

この基礎的なチュートリアルでは、AWS IoT Greengrass V2 の基本的な機能について知ることができます。このチュートリアルでは、以下の作業を行います。

1. Raspberry Piなどの Linux デバイスや Windows デバイスに AWS IoT Greengrass Core ソフトウェアをインストールして設定します。このデバイスは Greengrass コアデバイスです。
2. Greengrass コアデバイス上で、Hello World コンポーネントを開発します。コンポーネントは、Greengrass コアデバイス上で動作するソフトウェアです。
3. このコンポーネントを AWS クラウド にある AWS IoT Greengrass V2 にアップロードします。
4. このコンポーネントを、AWS クラウド から Greengrass コアデバイスにデプロイします。

> **Note**
>
> このチュートリアルでは、開発環境をセットアップする方法と、AWS IoT Greengrass の機能について説明します。実稼働デバイスをセットアップおよび設定する方法の詳細については、以下を参照してください。
>
> - AWS IoT Greengrass コアデバイスをセットアップする
> - AWS IoT Greengrass Core ソフトウェアをインストールします。

このチュートリアルは 20～30 分を要します。

トピック

前提条件

この入門チュートリアルを完了するには、以下が必要です。

- AWS アカウント。アカウントをお持ちでない場合は、「ステップ 1: AWS アカウントを設定する」を参照してください。
- AWS IoT Greengrass V2 をサポートする AWS リージョン を使用します。サポートされているリージョンのリストについては、「AWS 全般のリファレンス」の「AWS IoT Greengrass V2 のエンドポイントとクォータ」を参照してください。
- 管理者権限を持つ AWS Identity and Access Management (IAM) ユーザー。
- Greengrass コアデバイスとしてセットアップするデバイス。Raspberry Pi OS を搭載した Raspberry Pi (以前の Raspbian) または Windows 10 デバイスなど。このデバイスに対する管理者権限を持っているか、sudo を通してなどの方法で、管理者権限を取得できる必要があります。このデバイスにはインターネット接続が必要です。

  また、AWS IoT Greengrass Core ソフトウェアをインストールして実行するための要件を満たしている、別のデバイスを使用することもできます。詳細については、「サポートされているプラットフォームと要件」を参照してください。

  開発用コンピュータがこれらの要件を満たしている場合は、このチュートリアルで Greengrass コアデバイスとしてセットアップできます。
- デバイス上のすべてのユーザーに対して Python 3.5 以降がインストールされており、PATH 環境変数に追加されていること。Windows の場合は、すべてのユーザーに対して Python Launcher for Windows ランチャーがインストールされている必要があります。

  > **Important**
  >
  > Windows では、デフォルトでは Python がすべてのユーザーにインストールされません。Python をインストールするときには、インストールをカスタマイズして、AWS IoT Greengrass Core ソフトウェアが Python スクリプトを実行するように設定する必要があります。たとえば、グラフィカル Python インストーラを使用する場合には、次の操作を行います。
  >
  > 1. Install launcher for all users (recommended) (すべてのユーザーにランチャーをインストールする (推奨)) を選択します。
  > 2. Customize installation を選択します。
  > 3. を選択します。。Next

4. Install for all users を選択します。
5. Add Python to environment variables を選択します。
6. [Install] (インストール) を選択します。

詳細については、「Python 3 ドキュメント」の「Windows で Python を使用する」を参照してください。

- 開発コンピュータおよびデバイスに AWS Command Line Interface (AWS CLI) がインストールされており、認証情報が設定されていること。開発用コンピュータとデバイスで、同じ AWS リージョン リージョンを使って AWS CLI を設定しているか確認してください。AWS IoT Greengrass V2 を AWS CLI で使用するには、以下のいずれかのバージョン以降である必要があります。
  - 最小 AWS CLI V1 バージョン: v1.18.197
  - 最小 AWS CLI V2 バージョン: v2.1.11

> **Tip**
>
> 現在の AWS CLI のバージョンを確認するには、次のコマンドを実行します。
>
> ```
> aws --version
> ```

詳細については、「AWS Command Line Interface ユーザーガイド」の「AWS CLI のインストール、更新、アンインストール」と「AWS CLI の設定」を参照してください。

> **Note**
>
> 32 ビットのオペレーティングシステムを備えた Raspberry Pi などの、32 ビット ARM デバイスを使用する場合は、AWS CLI V1 をインストールします。AWS CLIV2 は 32 ビット ARM デバイスでは利用できません。詳細については、「AWS CLI バージョン 1 のインストール、更新、およびアンインストール」を参照してください。

ステップ 1: AWS アカウントを設定する

AWS アカウントへのサインアップ

AWS アカウントがない場合は、以下のステップを実行して作成します。

AWS アカウント にサインアップするには

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、通話呼び出しを受け取り、電話のキーパッドを使用して検証コードを入力するように求められます。

   AWS アカウントにサインアップすると、AWS アカウントのルートユーザーが作成されます。ルートユーザーには、アカウントのすべての AWS のサービスとリソースへのアクセス権があります。セキュリティのベストプラクティスとして、管理ユーザーに管理アクセスを割り当て、ルートユーザーアクセスが必要なタスクを実行する場合にのみ、ルートユーザーを使用してください。

サインアップ処理が完了すると、AWS からユーザーに確認メールが送信されます。https://aws.amazon.com/ の [アカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理ユーザーの作成

AWS アカウント にサインアップしたら、AWS アカウントのルートユーザー をセキュリティで保護し、AWS IAM Identity Center を有効にして、管理ユーザーを作成します。これにより、日常的なタスクにルートユーザーを使用しないようにします。

AWS アカウントのルートユーザーをセキュリティで保護する

1. [ルートユーザー] を選択し、AWS アカウント のメールアドレスを入力して、アカウント所有者として AWS Management Console にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、「AWS サインイン User Guide」の「Signing in as the root user」を参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

手順については、「IAM ユーザーガイド」の「AWS アカウント のルートユーザーの仮想 MFA デバイスを有効にする (コンソール)」を参照してください。

管理ユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Center の有効化」を参照してください。
2. IAM アイデンティティセンターで、管理ユーザーに管理者アクセスを付与します。

   IAM アイデンティティセンターディレクトリ をアイデンティティソースとして使用するチュートリアルについては、「AWS IAM Identity Center ユーザーガイド」の「デフォルトの IAM アイデンティティセンターディレクトリ でユーザーアクセスを設定する」を参照してください。

管理ユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM アイデンティティセンターのユーザーを使用してサインインする方法については、「AWS サインイン User Guide」の「Signing in to the AWS access portal」を参照してください。

ステップ2: 環境の構築

このセクションのステップに従って、AWS IoT Greengrass コアデバイスとして使用する Linux または Windows デバイスをセットアップします。

Linux デバイス (Raspberry Pi) をセットアップする

これらのステップでは、Raspberry Pi OS を備えた Raspberry Pi を使用していることを前提としています。別のデバイスまたはオペレーティングシステムを使用する場合は、お使いのデバイスに該当するドキュメントを参照してください。

AWS IoT Greengrass V2 の Raspberry Pi をセットアップするには

1. Raspberry Pi で SSH を有効にして、リモート接続します。詳細については、「Raspberry Pi ドキュメント」の「SSH (セキュアシェル)」を参照してください。

2. Raspberry Pi の IP アドレスを検索して SSH で接続します。これを行うには、Raspberry Pi 上で次のコマンドを実行します。

```
hostname -I
```

3. SSH で Raspberry Pi に接続します。

   開発用コンピュータに次のコマンドを実行します。*username* をサインインするユーザーの名前に置き換え、 を前のステップで見つけた IP アドレス*pi-ip-address*に置き換えます。

```
ssh username@pi-ip-address
```

> **Important**
>
> 開発コンピュータで Microsoft Windows の以前のバージョンを使用している場合、ssh コマンドがないか、ssh があっても Raspberry Pi に接続できない可能性があります。Raspberry Pi に接続するには、無料のオープンソース SSH クライアントである PuTTY をインストールすることができます。「PuTTY ドキュメント」を参照して、Raspberry Pi に接続します。

4. AWS IoT Greengrass Core ソフトウェアを実行するために必要な Java ランタイムをインストールします。Raspberry Pi で、次のコマンドを使用して Java 11 をインストールします。

```
sudo apt install default-jdk
```

   インストールが完了したら、次のコマンドを実行して Java が Raspberry Pi で実行されていることを確認します。

```
java -version
```

   このコマンドは、デバイス上で実行されている Java のバージョンを出力します。出力は、次の例のようになります。

```
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

ヒント: Raspberry Pi にカーネルパラメータを設定する

デバイスが Raspberry Pi の場合、次の手順を実行して Linux カーネルパラメータを表示と更新できます。

1. `/boot/cmdline.txt` ファイルを開きます。このファイルでは、Raspberry Pi の起動時に適用する Linux カーネルパラメータを指定します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを開きます。

   ```
   sudo nano /boot/cmdline.txt
   ```

2. `/boot/cmdline.txt` ファイルに次のカーネルパラメータが含まれていることを確認します。`systemd.unified_cgroup_hierarchy=0` パラメータは、cgroups v2 ではなく、cgroups v1 を使用することを指定します。

   ```
   cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
   ```

   `/boot/cmdline.txt` ファイルにこれらのパラメータが含まれていない場合、あるいは異なる値を持つこれらのパラメータが含まれている場合、これらのパラメータと値を含むようにファイルを更新してください。

3. `/boot/cmdline.txt` ファイルを更新した場合、Raspberry Pi を再起動して変更を適用します。

   ```
   sudo reboot
   ```

Linux デバイスをセットアップする (その他)

AWS IoT Greengrass V2 の Linux デバイスをセットアップするには

1. AWS IoT Greengrass Core ソフトウェアを実行するために必要な Java ランタイムをインストールします。Amazon Corretto または OpenJDK の長期サポートバージョンを使用することをお勧めします。バージョン 8 以降が必要です。次のコマンドは、デバイスに OpenJDK をインストールする方法を示しています。

   - Debian ベースまたは Ubuntu ベースのディストリビューションの場合:

```
sudo apt install default-jdk
```

- Red Hat ベースのディストリビューションの場合 :

```
sudo yum install java-11-openjdk-devel
```

- 複数 Amazon Linux 2:

```
sudo amazon-linux-extras install java-openjdk11
```

- 複数 Amazon Linux 2023:

```
sudo dnf install java-11-amazon-corretto -y
```

インストールが完了したら、次のコマンドを実行して Java が Linux デバイスで実行されていることを確認します。

```
java -version
```

このコマンドは、デバイス上で実行されている Java のバージョンを出力します。例えば、Debian ベースのディストリビューションでは、出力は次のサンプルのようになります。

```
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

2. (オプション) デバイスにコンポーネントを実行するデフォルトのシステムユーザーおよびグループを作成します。AWS IoT Greengrass Core ソフトウェアインストーラは、`--component-default-user` インストーラ引数でこのユーザーとグループを作成させるという選択もあります。詳細については、「インストーラ引数」を参照してください。

```
sudo useradd --system --create-home ggc_user
sudo groupadd --system ggc_group
```

3. AWS IoT Greengrass Core ソフトウェア (通常は `root`) を実行するユーザーが、sudo を任意のユーザーと任意のグループに実行する許可があることを確認してください。

   a. `/etc/sudoers` ファイルを開くには、次のコマンドを実行します。

```
sudo visudo
```

b. ユーザーの権限が次の例のようになっていることを確認します。

```
root    ALL=(ALL:ALL) ALL
```

4. (オプション) コンテナ化された Lambda 関数を実行するには、cgroups v1 を有効にし、メモリとデバイスの cgroups を有効にしてマウントする必要があります。コンテナ化された Lambda 関数を実行する予定がない場合、この手順を省略できます。

   これらの cgroups オプションを有効にするには、次の Linux カーネルパラメータを使用してデバイスを起動します。

   ```
   cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
   ```

   デバイスのカーネルパラメータを確認および設定するための情報については、オペレーティングシステムおよびブートローダーのドキュメントを参照してください。指示に従って、カーネルパラメータを永続的に設定します。

5. デバイスの要件 にある要件リストで示されているように、その他の必要となる依存関係をすべてデバイスにインストールします。

Windows デバイスをセットアップする

AWS IoT Greengrass V2 の Windows デバイスをセットアップするには

1. AWS IoT Greengrass Core ソフトウェアを実行するために必要な Java ランタイムをインストールします。Amazon Corretto または OpenJDK の長期サポートバージョンを使用することをお勧めします。バージョン 8 以降が必要です。
2. PATH システム変数で Java が使用可能か確認し、そうでない場合は追加します。 LocalSystem アカウントは AWS IoT Greengrass Core ソフトウェアを実行するため、ユーザーの PATH ユーザー変数ではなく、PATH システム変数に Java を追加する必要があります。以下の操作を実行します。

   a. Windows キーを押してスタートメニューを開きます。

   b. **environment variables** を入力して、スタートメニューからシステムオプションを検索します。

c. スタートメニューの検索結果から [Edit the system environment variables] (システム環境変数を編集) をクリックして、[System properties] (システムプロパティ) ウィンドウを開きます。

d. [Environment variables...] (環境変数...) を選択して、[Environment Variables] (環境可変) ウィンドウを開きます。

e. [System variables] (システム変数) で、[Path] (パス) 、[Edit] (編集) の順に選択します。[Edit environment variable] (環境変数の編集) ウィンドウでは、個別の行に各パスを表示できます。

f. Java インストールの `bin` フォルダへのパスが存在しているかを確認します。このパスは、次の例のように表示されます。

```
C:\\Program Files\\Amazon Corretto\\jdk11.0.13_8\\bin
```

g. [Path] (パス) で Java インストールの `bin` フォルダが で見つからない場合は、[New] (新規) を選択してこれを追加した上で、[OK] を選択します。

3. 管理者として Windows コマンドプロンプト `cmd.exe` を開きます。

4. Windows デバイスの LocalSystem アカウントにデフォルトユーザーを作成します。*#####*を安全なパスワードに置き換えます。

```
net user /add ggc_user password
```

> **Tip**
>
> Windows の構成によっては、ユーザーのパスワードの期限切れが、将来の日付に設定されている場合があります。Greengrass アプリケーションの動作を継続させるためには、パスワードの有効期限を追跡し、その期限が切れる前に更新します。ユーザーのパスワードには、期限切れを起こさないような設定も可能です。
>
> - ユーザーとパスワードの有効期限を確認するには、次のコマンドを実行します。
>
>   ```
>   net user ggc_user | findstr /C:expires
>   ```
>
> - ユーザーのパスワードが期限切れにならないように設定するには、次のコマンドを実行します。
>
>   ```
>   wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
>   ```

- wmic コマンドが廃止された Windows 10 以降を使用している場合は、次の PowerShell コマンドを実行します。

```
Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
 'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
```

5. デバイスに Microsoft から PsExec ユーティリティをダウンロードしてインストールします。
6. PsExec ユーティリティを使用して、デフォルトユーザーのユーザー名とパスワードを LocalSystem アカウントの認証情報マネージャーインスタンスに保存します。*#####*を以前に設定したユーザーのパスワードに置き換えます。

```
psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
```

PsExec License Agreement が開いたら、Accept を選択し、ライセンスに同意してコマンドを実行します。

> **Note**
>
> Windows デバイスでは、 LocalSystem アカウントが Greengrass nucleus を実行するため、 PsExec ユーティリティを使用してデフォルトのユーザー情報を LocalSystem アカウントに格納する必要があります。認証情報マネージャーアプリケーションを使用すると、この情報は アカウントではなく、現在ログオンしているユーザーの Windows LocalSystem アカウントに保存されます。

ステップ 3: AWS IoT Greengrass Core ソフトウェアをインストールする

このセクションの手順に従って、Raspberry Pi をローカル開発に使用できる AWS IoT Greengrass コアデバイスとして設定します。このセクションでは、以下を実行するインストーラをダウンロードして実行し、デバイスの AWS IoT Greengrass Core ソフトウェアを設定します。

- Greengrass nucleus コンポーネントをインストールします。nucleus は必須コンポーネントであり、デバイス上で AWS IoT Greengrass Core ソフトウェアを実行するための最小要件となります。詳細については、「Greengrass nucleus コンポーネント」を参照してください。

- デバイスを AWS IoT モノとして登録し、デバイスの AWS への接続を許可するデジタル証明書をダウンロードします。詳細については、「AWS IoT Greengrass のデバイス認証と認可」を参照してください。
- デバイスの AWS IoTモノ をモノグループに追加します。モノグループは、AWS IoT モノのグループまたはフリートです。モノグループを使用すると、Greengrass コアデバイスのフリートを管理できます。ソフトウェアコンポーネントをデバイスにデプロイするとき、個々のデバイスまたはデバイスのグループのどちらにデプロイするのかを選択することができます。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT でデバイスを管理する」を参照してください。
- Greengrass コアデバイスが AWS サービスとやり取りを許可する IAM ロールを作成します。デフォルトでは、このロールはデバイスが AWS IoT とやり取りし、Amazon CloudWatch Logs にログを送信することを許可します。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。
- AWS IoT Greengrass コマンドラインインターフェイス (`greengrass-cli`) をインストールします。これを使用することで、コアデバイスで開発するカスタムコンポーネントをテストすることができます。詳細については、「Greengrass コマンドラインインターフェイス」を参照してください。

AWS IoT Greengrass Core ソフトウェアをインストールする (コンソール)

1. AWS IoT Greengrass コンソールにサインインします。
2. [Get started with Greengrass] (Greengrass の使用を開始する) で、[Set up one core device] (1 つのコアデバイスをセットアップする) を選択します。
3. [Step 1: Register a Greengrass core device] (ステップ 1: Greengrass コアデバイスを登録する) にある [Core device name] (コアデバイス名) に、Greengrass コアデバイスの AWS IoT モノの名前を入力します。モノが存在しない場合、インストーラによって作成されます。
4. [Step 2: Add to a thing group to apply a continuous deployment] (ステップ 2: モノグループに追加して継続的デプロイを適用する)、に対して [Thing group] (モノのグループ) で、AWS IoT コアデバイスを追加するモノグループ。

   - [Enter a new group name] (新しいグループ名を入力) を選択した場合、[Thing group name] (モノグループ名) には作成する新しいグループの名前を入力します。インストーラによって新しいグループが作成されます。
   - [Select an existing group] (既存のグループを選択) を選択した場合、[Thing group name] (モノグループ名)には使用する既存のグループを選択します。

- [No group] (グループなし) を選択した場合、インストーラはコアデバイスをモノグループに追加しません。

5. [Step 3: Install the Greengrass Core software] (ステップ 3: Greengrass Coreソフトウェアをインストールする) で、以下のステップを実行します。

   a. コアデバイスのオペレーティングシステムを選択します。Linux または Windows のいずれかです。

   b. デバイスに AWS 認証情報を提供して、インストーラがコアデバイスの AWS IoT と IAM リソースをプロビジョニングできるようにします。セキュリティを強化するには、プロビジョニングに必要な最小限の許可のみを与える IAM ロールの一時的な認証情報を取得することをお勧めします。詳細については、「インストーラがリソースをプロビジョニングするための最小限の IAM ポリシー」を参照してください。

      > **Note**
      >
      > インストーラが認証情報を保存したり保管することはありません。

      デバイスに、次のいずれかを実行して、認証情報を取得して AWS IoT Greengrass Core ソフトウェアのインストーラを利用できるようにしてください。

      - (推奨) からの一時認証情報を使用する AWS IAM Identity Center

        i. IAM Identity Center からアクセスキー ID、シークレットアクセスキー、およびセッショントークンを指定します。詳細については、IAM Identity Center ユーザーガイドの「一時的な認証情報の取得と更新」の「認証情報の手動更新」を参照してください。

        ii. 次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアに認証情報を提供します。

        Linux or Unix

        ```
        export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
        export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/
        bPxRfiCYEXAMPLEKEY
        export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
        ```

Windows Command Prompt (CMD)

```
set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
set AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

PowerShell

```
$env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
$env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/
bPxRfiCYEXAMPLEKEY"
$env:AWS_SESSION_TOKEN="AQoDYXdzEJr1K...o5OytwEXAMPLE="
```

- IAM ロールから一時的なセキュリティ認証情報を使用します。

  i. 継承する IAM ロールから、アクセスキー ID、シークレットアクセスキー、セッショントークンを提供します。これらの認証情報を取得する方法の詳細については、「IAM ユーザーガイド」の「一時的なセキュリティ認証情報のリクエスト」を参照してください。

  ii. 次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアに認証情報を提供します。

  Linux or Unix

  ```
  export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
  export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/
  bPxRfiCYEXAMPLEKEY
  export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
  ```

  Windows Command Prompt (CMD)

  ```
  set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
  set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
  set AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
  ```

  PowerShell

  ```
  $env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
  ```

```
$env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/
bPxRfiCYEXAMPLEKEY"
$env:AWS_SESSION_TOKEN="AQoDYXdzEJr1K...o5OytwEXAMPLE="
```

- IAM ユーザーからの長期的な認証情報を使用する:

  i. IAM ユーザーのアクセスキー ID とシークレットアクセスキーを提供します。後で削除するプロビジョニング用の IAM ユーザーを作成できます。ユーザーに付与する IAM ポリシーについては、「」を参照してくださいインストーラがリソースをプロビジョニングするための最小限の IAM ポリシー。長期認証情報を取得する方法の詳細については、「IAM ユーザーガイド」の「IAM ユーザーのアクセスキー管理」を参照してください。

  ii. 次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアに認証情報を提供します。

  Linux or Unix

  ```
  export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
  export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/
  bPxRfiCYEXAMPLEKEY
  ```

  Windows Command Prompt (CMD)

  ```
  set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
  set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
  ```

  PowerShell

  ```
  $env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
  $env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/
  bPxRfiCYEXAMPLEKEY"
  ```

  iii. （オプション) Greengrass デバイスをプロビジョニングする IAM ユーザーを作成した場合は、そのユーザーを削除します。

  iv. （オプション) 既存の IAM ユーザーのアクセスキー ID とシークレットアクセスキーを使用した場合は、そのユーザーのキーを更新して無効になるようにします。詳細については、「 ユーザーガイド」の「アクセスキーの更新」を参照してください。 AWS Identity and Access Management

c. [Run the installer] (インストーラの実行) で、以下のステップを実行します。

i. [Download the installer] (インストーラをダウンロードする) で、[Copy] (コピー) を選択して、コアデバイス上でコピーしたコマンドを実行します。このコマンドは、AWS IoT Greengrass Core ソフトウェアの最新バージョンをダウンロードし、デバイスで解凍します。

ii. [Run installer] (インストーラを実行する) で、[Copy] (コピー) を選択して、コアデバイス上でコピーしたコマンドを実行します。このコマンドでは、AWS IoT Greengrass Core ソフトウェアインストーラを実行し、コアデバイスの AWS リソースを設定するために、前のステップで指定した AWS IoT モノとモノグループ名を使用します。

   また、このコマンドは次を行います。

   - AWS IoT Greengrass Core ソフトウェアを、ブート時に実行されるシステムサービスとして設定します。Linux デバイスでは、これは Systemd init システムが必要です。

     > **Important**
     >
     > Windows コアデバイスでは、AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります

   - AWS IoT Greengrass CLI コンポーネントをデプロイします。このコンポーネントは、コアデバイスでカスタム Greengrass コンポーネントを開発できるようにするためのコマンドラインツールです。
   - コアデバイスでソフトウェアコンポーネントを実行するために `ggc_user` システムユーザーを使用するように指定します。Linux デバイスでは、このコマンドも `ggc_group` システムグループを使用するように指定し、さらにインストーラによってシステムユーザーとグループが、ユーザーに代わって作成されます。

   このコマンドを実行すると、インストーラが成功したことを示す次のメッセージが表示されます。

   ```
   Successfully configured Nucleus with provisioned resource details!
   Configured Nucleus to deploy aws.greengrass.Cli component
   Successfully set up Nucleus as a system service
   ```

> **Note**
>
> Linux デバイスがあるものの、systemd がない場合には、インストーラはソフトウェアをシステムサービスとして設定せず、nucleus をシステムサービスとして設定できたことを示す成功メッセージは表示されません。

AWS IoT Greengrass Core ソフトウェアをインストールする (CLI)

AWS IoT Greengrass Core ソフトウェアをインストールして設定するには

1. Greengrass コアデバイスで、次のコマンドを実行してホームディレクトリに切り替えます。

   Linux or Unix

   ```
   cd ~
   ```

   Windows Command Prompt (CMD)

   ```
   cd %USERPROFILE%
   ```

   PowerShell

   ```
   cd ~
   ```

2. コアデバイス上で、AWS IoT Greengrass Core ソフトウェアを `greengrass-nucleus-latest.zip` という名前のファイルにダウンロードします。

   Linux or Unix

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip > greengrass-nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip > greengrass-nucleus-latest.zip
   ```

PowerShell

```
iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip -OutFile greengrass-nucleus-latest.zip
```

このソフトウェアをダウンロードすると、Greengrass Core ソフトウェアのライセンス契約に同意したものと見なされます。

3. AWS IoT Greengrass Core ソフトウェアをデバイス上のフォルダに解凍します。を使用するフォルダ*GreengrassInstaller*に置き換えます。

   Linux or Unix

   ```
   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-
   nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir GreengrassInstaller && tar -xf greengrass-nucleus-latest.zip -
   C GreengrassInstaller && del greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   Expand-Archive -Path greengrass-nucleus-latest.zip -DestinationPath .\
   \GreengrassInstaller
   rm greengrass-nucleus-latest.zip
   ```

4. 次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアのインストーラを起動します。このコマンドは次のことを行います。

   - コアデバイスの動作に必要な AWS リソースを作成します。
   - AWS IoT Greengrass Core ソフトウェアを、ブート時に実行されるシステムサービスとして設定します。Linux デバイスでは、これは Systemd init システムが必要です。

   > **Important**
   >
   > Windows コアデバイスでは、AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります

- AWS IoT Greengrass CLI コンポーネントをデプロイします。このコンポーネントは、コアデバイスでカスタム Greengrass コンポーネントを開発できるようにするためのコマンドラインツールです。
- コアデバイスでソフトウェアコンポーネントを実行するために `ggc_user` システムユーザーを使用するように指定します。Linux デバイスでは、このコマンドも `ggc_group` システムグループを使用するように指定し、さらにインストーラによってシステムユーザーとグループが、ユーザーに代わって作成されます。

コマンドの引数値を次のように置き換えます。

a. */greengrass/v2* または *C:\greengrass\v2*: AWS IoT Greengrass Core ソフトウェアのインストールに使用するルートフォルダへのパス。

b. *GreengrassInstaller*。AWS IoT Greengrass Core ソフトウェアのインストーラを展開したフォルダへのパス。

c. *#####*。リソースを検索または作成する AWS リージョン。

d. *MyGreengrassCore*。Greengrass コアデバイスの AWS IoT モノの名前。モノが存在しない場合、インストーラによって作成されます。インストーラは、証明書をダウンロードして AWS IoT モノとして認証します。詳細については、「AWS IoT Greengrass のデバイス認証と認可」を参照してください。

> **Note**
>
> モノの名前にコロン (:) 記号を含むことができません。

e. *MyGreengrassCoreGroup*。Greengrass コアデバイスの AWS IoT モノグループの名前。モノグループが存在しない場合、インストーラはそのグループを作成してモノを追加します。モノグループが存在してアクティブなデプロイがある場合、コアデバイスはデプロイで指定されたソフトウェアをダウンロードして実行します。

> **Note**
>
> モノグループ名にコロン (:) 記号を含めることはできません。

f. *GreengrassV2IoTThingPolicy#*Greengrass コアデバイスが AWS IoT および AWS IoT Greengrass と通信できるようにする AWS IoT ポリシーの名前。AWS IoT ポリシーが存在しない場合、インストーラが許可を与える AWS IoT ポリシーをこの名前で作成します。

ユースケースに合わせて、このポリシーのアクセス許可を制限することができます。詳細については、「AWS IoT Greengrass V2 コアデバイス向けの最低限の AWS IoT ポリシー」を参照してください。

g. *GreengrassV2TokenExchangeRole*。Greengrass コアデバイスが一時的に AWS 認証情報を取得できるようにする IAM ロールの名前。ロールが存在しない場合、インストーラがロールを作成し、*GreengrassV2TokenExchangeRole*Access という名前のポリシーを作成してアタッチします。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

h. *GreengrassCoreTokenExchangeRoleAlias*。Greengrass コアデバイスが後で一時的な認証情報を取得できるようにする IAM ロールのエイリアス。ロールエイリアスが存在しない場合、インストーラがロールエイリアスを作成し、指定した IAM ロールを指します。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

Linux or Unix

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
  -jar ./GreengrassInstaller/lib/Greengrass.jar \
  --aws-region region \
  --thing-name MyGreengrassCore \
  --thing-group-name MyGreengrassCoreGroup \
  --thing-policy-name GreengrassV2IoTThingPolicy \
  --tes-role-name GreengrassV2TokenExchangeRole \
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias \
  --component-default-user ggc_user:ggc_group \
  --provision true \
  --setup-system-service true \
  --deploy-dev-tools true
```

Windows Command Prompt (CMD)

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" ^
  -jar ./GreengrassInstaller/lib/Greengrass.jar ^
  --aws-region region ^
  --thing-name MyGreengrassCore ^
  --thing-group-name MyGreengrassCoreGroup ^
  --thing-policy-name GreengrassV2IoTThingPolicy ^
  --tes-role-name GreengrassV2TokenExchangeRole ^
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias ^
```

```
  --component-default-user ggc_user ^
  --provision true ^
  --setup-system-service true ^
  --deploy-dev-tools true
```

PowerShell

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" `
  -jar ./GreengrassInstaller/lib/Greengrass.jar `
  --aws-region region `
  --thing-name MyGreengrassCore `
  --thing-group-name MyGreengrassCoreGroup `
  --thing-policy-name GreengrassV2IoTThingPolicy `
  --tes-role-name GreengrassV2TokenExchangeRole `
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias `
  --component-default-user ggc_user `
  --provision true `
  --setup-system-service true `
  --deploy-dev-tools true
```

> **Note**
>
> メモリが制限されているデバイスで AWS IoT Greengrass を実行する場合、AWS IoT Greengrass Core ソフトウェアが使用するメモリ量を制御できます。メモリ割り当てを制御するには、nucleus コンポーネントの `jvmOptions` 設定パラメータで JVM ヒープのサイズオプションを設定できます。詳細については、「JVM オプションでメモリ割り当てを制御する」を参照してください。

このコマンドを実行すると、インストーラが成功したことを示す次のメッセージが表示されます。

```
Successfully configured Nucleus with provisioned resource details!
Configured Nucleus to deploy aws.greengrass.Cli component
Successfully set up Nucleus as a system service
```

> **Note**
>
> Linux デバイスがあるものの、systemd がない場合には、インストーラはソフトウェアをシステムサービスとして設定せず、nucleus をシステムサービスとして設定できたことを示す成功メッセージは表示されません。

(オプション) Greengrass ソフトウェアを実行する (Linux)

ソフトウェアをシステムサービスとしてインストールした場合、インストーラがソフトウェアを実行します。実行されない場合は、ソフトウェアを手動で実行する必要があります。インストーラがソフトウェアをシステムサービスとして設定しているかどうかを確認するには、インストーラの出力に次の行がないか探します。

```
Successfully set up Nucleus as a system service
```

このメッセージが表示されていない場合は、次の手順に従ってソフトウェアを実行します。

1. 次のコマンドを使用してソフトウェアを起動します。

   ```
   sudo /greengrass/v2/alts/current/distro/bin/loader
   ```

   正常に起動すると、ソフトウェアは次のメッセージを出力します。

   ```
   Launched Nucleus successfully.
   ```

2. AWS IoT Greengrass Core ソフトウェアを実行したままの状態にしておくため、現在のコマンドシェルは開いたままにしておく必要があります。SSH を使用してコアデバイスに接続する場合は、開発コンピュータで次のコマンドを実行して 2 番目の SSH セッションを開き、コアデバイス上で別のコマンドを実行するために使用することができます。*username* をサインインするユーザーの名前に置き換え、 をデバイスの IP アドレス*pi-ip-address*に置き換えます。

   ```
   ssh username@pi-ip-address
   ```

Greengrass システムサービスとやり取りする方法の詳細については、「Greengrass nucleus をシステムサービスとして設定する」を参照してください。

デバイス上の Greengrass CLI のインストールを確認する

Greengrass CLI のデプロイには、最大で 1 分かかります。次のコマンドを実行すると、デプロイのステータスを確認できます。をコアデバイスの名前*MyGreengrassCore*に置き換えます。

```
aws greengrassv2 list-effective-deployments --core-device-thing-name MyGreengrassCore
```

`coreDeviceExecutionStatus` はコアデバイスへのデプロイのステータスを示しています。ステータスが `SUCCEEDED` のときに次のコマンドを実行して、Greengrass CLI がインストールされ、実行されていることを確認します。をルートフォルダへのパス*/greengrass/v2*に置き換えます。

Linux or Unix

```
/greengrass/v2/bin/greengrass-cli help
```

Windows Command Prompt (CMD)

```
C:\greengrass\v2\bin\greengrass-cli help
```

PowerShell

```
C:\greengrass\v2\bin\greengrass-cli help
```

このコマンドは Greengrass CLI に関するヘルプ情報を出力します。`greengrass-cli` が見つからない場合は、デプロイで Greengrass CLI のインストールに失敗した可能性があります。詳細については、「トラブルシューティング AWS IoT Greengrass V2」を参照してください。

次のコマンドを実行して、 AWS IoT Greengrass CLI をデバイスに手動でデプロイすることもできます。

- *region* AWS リージョン は、使用する に置き換えます。 AWS CLI デバイスで の設定に使用した AWS リージョン のと同じ を使用していることを確認してください。
- *account-id* を ID に置き換えます AWS アカウント 。
- をコアデバイスの名前*MyGreengrassCore*に置き換えます。

Linux, macOS, or Unix

```
aws greengrassv2 create-deployment \
```

```
  --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" \
  --components '{
    "aws.greengrass.Cli": {
      "componentVersion": "2.12.2"
    }
  }'
```

Windows Command Prompt (CMD)

```
aws greengrassv2 create-deployment ^
  --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" ^
  --components "{\"aws.greengrass.Cli\":{\"componentVersion\":\"2.12.2\"}}"
```

PowerShell

```
aws greengrassv2 create-deployment `
  --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" `
  --components '{\"aws.greengrass.Cli\":{\"componentVersion\":\"2.12.2\"}}'
```

> **Tip**
>
> PATH 環境変数に */greengrass/v2*/bin (Linux) または *C:\greengrass\v2*\bin (Windows) を追加すると、絶対パスなしで `greengrass-cli` を実行することができます。

AWS IoT Greengrass Core ソフトウェアとローカル開発ツールは、デバイス上で実行されます。次に、デバイスで Hello Worldの AWS IoT Greengrass コンポーネントを開発することができます。

ステップ 4: デバイス上でコンポーネントを開発およびテストする

コンポーネントは、 AWS IoT Greengrass コアデバイスで実行されるソフトウェアモジュールです。コンポーネントを使用すると、複雑なアプリケーションを個別のビルディングブロックとして作成し管理することができ、Greengrass コアデバイス間で再利用することができます。すべてのコンポーネントは、recipe とアーティファクトで設定されます。

- recipe

  すべてのコンポーネントには、メタデータを定義する recipe ファイルが含まれています。recipe では、コンポーネントの設定パラメータ、コンポーネントの依存関係、ライフサイクル、プラッ

トフォームの互換性も指定します。コンポーネントのライフサイクルは、コンポーネントのインストール、実行、およびシャットダウンを行うコマンドを定義します。詳細については、「AWS IoT Greengrass コンポーネントレシピのリファレンス」を参照してください。

recipe は JSON または YAML 形式で定義できます。

- アーティファクト

  コンポーネントは、コンポーネントバイナリであるアーティファクトを必要な数だけ持つことができます。アーティファクトには、スクリプト、コンパイルされたコード、静的リソース、およびコンポーネントが消費するその他のファイルが含まれます。コンポーネントはコンポーネントの依存関係からアーティファクトを消費することもできます。

では AWS IoT Greengrass、Greengrass CLI を使用して、 AWS クラウドとやり取りすることなく、Greengrass コアデバイスでコンポーネントをローカルで開発およびテストできます。ローカルコンポーネントを完了すると、コンポーネント recipe とアーティファクトを使用して AWS クラウド AWS IoT Greengrass のサービスにそのコンポーネントを作成し、すべての Greengrass コアデバイスにデプロイできます。コンポーネントの詳細については、「AWS IoT Greengrass コンポーネントを開発する」を参照してください。

このセクションでは、基本的な Hello World コンポーネントをコアデバイス上でローカルに作成して実行する方法を説明します。

デバイスで Hello World コンポーネントを開発するには

1. recipe とアーティファクトのサブフォルダを含むコンポーネントのフォルダを作成します。Greengrass コアデバイスで次のコマンドを実行してこれらのフォルダを作成し、コンポーネントフォルダに変更します。*~/greengrassv2* または *%USERPROFILE%\greengrassv2* をローカル開発に使用するフォルダへのパスに置き換えます。

   Linux or Unix

   ```
   mkdir -p ~/greengrassv2/{recipes,artifacts}
   cd ~/greengrassv2
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir %USERPROFILE%\greengrassv2\\recipes, %USERPROFILE%\greengrassv2\\artifacts
   cd %USERPROFILE%\greengrassv2
   ```

PowerShell

```
mkdir ~/greengrassv2/recipes, ~/greengrassv2/artifacts
cd ~/greengrassv2
```

2. テキストエディタを使用して、コンポーネントのメタデータ、パラメータ、依存関係、ライフサイクル、プラットフォーム機能を定義する recipe ファイルを作成します。recipe ファイル名にコンポーネントのバージョンを含めるようにして、どの recipe がどのコンポーネントバージョンを反映しているのかを特定できるようにします。recipe には YAML 形式または JSON 形式を選択できます。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

   JSON

   ```
   nano recipes/com.example.HelloWorld-1.0.0.json
   ```

   YAML

   ```
   nano recipes/com.example.HelloWorld-1.0.0.yaml
   ```

   > **Note**
   >
   > AWS IoT Greengrass はコンポーネントのセマンティックバージョンを使用します。セマンティックバージョンは、major.minor.patch といった番号システムに準拠します。例えば、バージョン `1.0.0` は、コンポーネントの最初のメジャーリリースを表しています。詳細については、「セマンティックバージョンの仕様」を参照してください。

3. ファイルに次の recipe を貼り付けます。

   JSON

   ```
   {
     "RecipeFormatVersion": "2020-01-25",
     "ComponentName": "com.example.HelloWorld",
     "ComponentVersion": "1.0.0",
     "ComponentDescription": "My first AWS IoT Greengrass component.",
   ```

```
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "world"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "run": "python3 -u {artifacts:path}/hello_world.py {configuration:/
Message}"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "run": "py -3 -u {artifacts:path}/hello_world.py {configuration:/
Message}"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: My first AWS IoT Greengrass component.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    Message: world
Manifests:
  - Platform:
      os: linux
    Lifecycle:
```

```
    run: |
      python3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
- Platform:
    os: windows
  Lifecycle:
    run: |
      py -3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
```

この recipe の ComponentConfiguration セクションはパラメータである Message を定義しており、デフォルトでは world になります。Manifests セクションはマニフェストを定義しており、これは、プラットフォームのライフサイクル指示とアーティファクトのセットです。たとえば、複数のマニフェストを定義して、さまざまなプラットフォームに対して異なるインストール手順を指定することができます。マニフェストでは、Lifecycle セクションは、Greengrass コアデバイスに Message パラメータ値を引数として Hello World スクリプトを実行するように指示します。

4. 次のコマンドを実行して、コンポーネントアーティファクト用のフォルダを作成します。

   Linux or Unix

   ```
   mkdir -p artifacts/com.example.HelloWorld/1.0.0
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir artifacts\com.example.HelloWorld\1.0.0
   ```

   PowerShell

   ```
   mkdir artifacts\com.example.HelloWorld\1.0.0
   ```

   Important

   アーティファクトフォルダのパスには、次のフォーマットを使用する必要があります。recipe で指定したコンポーネント名とバージョンを含めてください。

   ```
   artifacts/componentName/componentVersion/
   ```

5. テキストエディタを使用して、Hello World コンポーネントの Python スクリプトアーティファクトファイルを作成します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

   ```
   nano artifacts/com.example.HelloWorld/1.0.0/hello_world.py
   ```

   次の Python スクリプトをコピーしてファイルに貼り付けます。

   ```
   import sys

   message = "Hello, %s!" % sys.argv[1]

   # Print the message to stdout, which Greengrass saves in a log file.
   print(message)
   ```

6. ローカル AWS IoT Greengrass CLI を使用して、Greengrass コアデバイスのコンポーネントを管理します。

   次のコマンドを実行して、コンポーネントを AWS IoT Greengrass コアにデプロイします。*/greengrass/v2* または *C:\greengrass\v2* を AWS IoT Greengrass V2 ルートフォルダに置き換え、*~/greengrassv2* または *%USERPROFILE%\greengrassv2* をコンポーネント開発フォルダに置き換えます。

   Linux or Unix

   ```
   sudo /greengrass/v2/bin/greengrass-cli deployment create \
     --recipeDir ~/greengrassv2/recipes \
     --artifactDir ~/greengrassv2/artifacts \
     --merge "com.example.HelloWorld=1.0.0"
   ```

   Windows Command Prompt (CMD)

   ```
   C:\greengrass\v2\bin\greengrass-cli deployment create ^
     --recipeDir %USERPROFILE%\greengrassv2\recipes ^
     --artifactDir %USERPROFILE%\greengrassv2\artifacts ^
     --merge "com.example.HelloWorld=1.0.0"
   ```

PowerShell

```
C:\greengrass\v2\bin\greengrass-cli deployment create `
  --recipeDir ~/greengrassv2/recipes `
  --artifactDir ~/greengrassv2/artifacts `
  --merge "com.example.HelloWorld=1.0.0"
```

このコマンドは、`recipes` では recipe 、`artifacts` では Python スクリプトを使用するコンポーネントを追加します。`--merge` オプションは指定したコンポーネントとバージョンを追加または更新します。

7. AWS IoT Greengrass Core ソフトウェアは、コンポーネントプロセスからの標準出力を `logs`フォルダのログファイルに保存します。Hello World コンポーネントが実行され、メッセージが表示されることを確認するには、次のコマンドを実行します。

   Linux or Unix

   ```
   sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
   ```

   Windows Command Prompt (CMD)

   ```
   type C:\greengrass\v2\logs\com.example.HelloWorld.log
   ```

   `type` コマンドは、ファイルのコンテンツを端末に書き込みます。このコマンドを複数回実行して、ファイル内の変更を確認してください。

   PowerShell

   ```
   gc C:\greengrass\v2\logs\com.example.HelloWorld.log -Tail 10 -Wait
   ```

   次の例に示すようなメッセージが表示されます。

   ```
   Hello, world!
   ```

> **Note**
>
> ファイルが存在しない場合、ローカルデプロイがまだ完了していない可能性があります。ファイルが 15 秒以内に表示されない場合は、デプロイが失敗している可能性があります。これは、recipe が有効でない場合などに発生します。次のコマンドを実行して、 AWS IoT Greengrass コアログファイルを表示します。このファイルは、Greengrass コアデバイスのデプロイサービスからのログが含まれます。
>
> Linux or Unix
>
> ```
> sudo tail -f /greengrass/v2/logs/greengrass.log
> ```
>
> Windows Command Prompt (CMD)
>
> ```
> type C:\greengrass\v2\logs\greengrass.log
> ```
>
> type コマンドは、ファイルのコンテンツを端末に書き込みます。このコマンドを複数回実行して、ファイル内の変更を確認してください。
>
> PowerShell
>
> ```
> gc C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
> ```

8. ローカルコンポーネントを変更して、コードを繰り返してテストします。テキストエディタhello_world.pyで を開き、4 行目に次のコードを追加して、 AWS IoT Greengrass コアがログに記録するメッセージを編集します。

```
message += " Greetings from your first Greengrass component."
```

hello_world.py スクリプトには次の内容が表示されるはずです。

```
import sys

message = "Hello, %s!" % sys.argv[1]
message += " Greetings from your first Greengrass component."

# Print the message to stdout, which Greengrass saves in a log file.
print(message)
```

9. 次のコマンドを実行して、変更内容でコンポーネントを更新します。

   Linux or Unix

   ```
   sudo /greengrass/v2/bin/greengrass-cli deployment create \
     --recipeDir ~/greengrassv2/recipes \
     --artifactDir ~/greengrassv2/artifacts \
     --merge "com.example.HelloWorld=1.0.0"
   ```

   Windows Command Prompt (CMD)

   ```
   C:\greengrass\v2\bin\greengrass-cli deployment create ^
     --recipeDir %USERPROFILE%\greengrassv2\recipes ^
     --artifactDir %USERPROFILE%\greengrassv2\artifacts ^
     --merge "com.example.HelloWorld=1.0.0"
   ```

   PowerShell

   ```
   C:\greengrass\v2\bin\greengrass-cli deployment create `
     --recipeDir ~/greengrassv2/recipes `
     --artifactDir ~/greengrassv2/artifacts `
     --merge "com.example.HelloWorld=1.0.0"
   ```

   このコマンドは、最新の Hello World アーティファクトで `com.example.HelloWorld` コンポーネントを更新します。

10. 次のコマンドを実行して、コンポーネントを再起動します。コンポーネントを再起動すると、コアデバイスは最新の変更を使用します。

    Linux or Unix

    ```
    sudo /greengrass/v2/bin/greengrass-cli component restart \
      --names "com.example.HelloWorld"
    ```

    Windows Command Prompt (CMD)

    ```
    C:\greengrass\v2\bin\greengrass-cli component restart ^
      --names "com.example.HelloWorld"
    ```

PowerShell

```
C:\greengrass\v2\bin\greengrass-cli component restart `
  --names "com.example.HelloWorld"
```

11. ログをもう一度確認して、Hello World コンポーネントが新しいメッセージを出力することか確認してください。

    Linux or Unix

    ```
    sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
    ```

    Windows Command Prompt (CMD)

    ```
    type C:\greengrass\v2\logs\com.example.HelloWorld.log
    ```

    type コマンドは、ファイルのコンテンツを端末に書き込みます。このコマンドを複数回実行して、ファイル内の変更を確認してください。

    PowerShell

    ```
    gc C:\greengrass\v2\logs\com.example.HelloWorld.log -Tail 10 -Wait
    ```

    次の例に示すようなメッセージが表示されます。

    ```
    Hello, world! Greetings from your first Greengrass component.
    ```

12. コンポーネントの設定パラメータを更新して、さまざまな設定をテストできます。コンポーネントをデプロイするときに、設定の更新を指定して、コアデバイス上のコンポーネントの設定を変更する方法を定義することができます。デフォルト値にリセットする設定値と、コアデバイスにマージする新しい設定値を指定することができます。詳細については、「コンポーネント設定の更新」を参照してください。

    以下の操作を実行します。

    a. テキストエディタを使用して、hello-world-config-update.json という名前のファイルを作成し、設定の更新を含めます。

例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano hello-world-config-update.json
```

b. 次の JSON オブジェクトをファイルにコピーして貼り付けます。この JSON オブジェクトは、`friend` の値を `Message` パラメータにマージして、その値を更新する設定更新を定義しています。この設定更新では、リセットする値は指定されていません。マージの更新によって既存の値が置き換えられるため、`Message` パラメータをリセットする必要はありません。

```
{
  "com.example.HelloWorld": {
    "MERGE": {
      "Message": "friend"
    }
  }
}
```

c. 次のコマンドを実行して、設定の更新を Hello World コンポーネントにデプロイします。

Linux or Unix

```
sudo /greengrass/v2/bin/greengrass-cli deployment create \
  --merge "com.example.HelloWorld=1.0.0" \
  --update-config hello-world-config-update.json
```

Windows Command Prompt (CMD)

```
C:\greengrass\v2\bin\greengrass-cli deployment create ^
  --merge "com.example.HelloWorld=1.0.0" ^
  --update-config hello-world-config-update.json
```

PowerShell

```
C:\greengrass\v2\bin\greengrass-cli deployment create `
  --merge "com.example.HelloWorld=1.0.0" `
  --update-config hello-world-config-update.json
```

d. ログをもう一度確認して、Hello World コンポーネントが新しいメッセージを出力することか確認してください。

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\logs\com.example.HelloWorld.log
```

type コマンドは、ファイルのコンテンツを端末に書き込みます。このコマンドを複数回実行して、ファイル内の変更を確認してください。

PowerShell

```
gc C:\greengrass\v2\logs\com.example.HelloWorld.log -Tail 10 -Wait
```

次の例に示すようなメッセージが表示されます。

```
Hello, friend! Greetings from your first Greengrass component.
```

13. コンポーネントのテストが完了したら、コアデバイスから削除します。以下のコマンドを実行します。

Linux or Unix

```
sudo /greengrass/v2/bin/greengrass-cli deployment create --
remove="com.example.HelloWorld"
```

Windows Command Prompt (CMD)

```
C:\greengrass\v2\bin\greengrass-cli deployment create --
remove="com.example.HelloWorld"
```

PowerShell

```
C:\greengrass\v2\bin\greengrass-cli deployment create --
remove="com.example.HelloWorld"
```

> **Important**
>
> この手順は、コンポーネントを AWS IoT Greengrass にアップロードした後、コアデバイスにコンポーネントをデプロイし直すために必要です。そうしなかった場合、ローカルデプロイはコンポーネントの異なるバージョンを指定しているため、デプロイがバージョンの互換性エラーで失敗します。

次のコマンドを実行して、`com.example.HelloWorld` コンポーネントが、デバイス上のコンポーネントリストに表示されないことを確認します。

Linux or Unix

```
sudo /greengrass/v2/bin/greengrass-cli component list
```

Windows Command Prompt (CMD)

```
C:\greengrass\v2\bin\greengrass-cli component list
```

PowerShell

```
C:\greengrass\v2\bin\greengrass-cli component list
```

Hello World コンポーネントは完了し、 AWS IoT Greengrass クラウドサービスにアップロードできるようになりました。その後に、コンポーネントを Greengrass コアデバイスにデプロイできます。

ステップ 5: AWS IoT Greengrass サービスでコンポーネントを作成する

コアデバイスでのコンポーネントの開発が完了したら、コンポーネントを AWS クラウド の AWS IoT Greengrass サービスにアップロードすることができます。また、コンポーネントをAWS IoT Greengrass コンソールで直接作成することもできます。AWS IoT Greengrass はコンポーネントをホストするコンポーネント管理サービスを提供するため、個々のデバイスやデバイスのフリートにデプロイすることができます。コンポーネントを AWS IoT Greengrass サービスにアップロードするには、以下のステップを実行します。

- コンポーネントのアーティファクトを S3 バケットにのアップロードします。
- 各アーティファクトの Amazon Simple Storage Service (Amazon S3) URI をコンポーネント recipe に追加します。
- コンポーネント recipe から、AWS IoT Greengrass にコンポーネントを作成します。

このセクションでは、Greengrass コアデバイスでこれらの手順を実行して、Hello World コンポーネントを AWS IoT Greengrass サービスにアップロードします。

AWS IoT Greengrass でコンポーネントを作成する (コンソール)

1. AWS アカウントの S3 バケットを使用して、AWS IoT Greengrass コンポーネントアーティファクトをホストします。コンポーネントをコアデバイスにデプロイすると、デバイスがバケットからコンポーネントのアーティファクトをダウンロードします。

   既存の S3 バケットを使用するか、新しいバケットを作成することができます。

   a. Amazon S3 コンソールで [Buckets] (バケット)、[Create bucket] (バケットの作成) の順に選択します。

   b. [Bucket name] (バケット名) に、一意のバケット名前を入力します。たとえば、**`greengrass-component-artifacts-`*`region`*`-`*`123456789012`*** を使用できます。*`123456789012`* は自分の AWS アカウント ID に置き換え、*`region`* はこのチュートリアルで使用する AWS リージョン に置き換えます。

   c. [AWS リージョン] で、このチュートリアルで使用する AWS リージョンを選択します。

   d. [Create bucket] (バケットの作成) を選択します。

   e. [Buckets] (バケット) で作成したバケットを選択し、`hello_world.py` スクリプトをバケット内の `artifacts/com.example.HelloWorld/1.0.0` フォルダにアップロードし

ます。オブジェクトを S3 バケットにアップロードする際の詳細については、「Amazon Simple Storage Service ユーザーガイド」の「オブジェクトをアップロードする」を参照してください。

f. S3 バケット内にある `hello_world.py` オブジェクトの S3 URI をコピーします。この URI は次の例のようになります。*DOC-EXAMPLE-BUCKET* を S3 バケットの名前に置き換えます。

```
s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/hello_world.py
```

2. コアデバイスが S3 バケット内のコンポーネントアーティファクトにアクセスすることを許可します。

各コアデバイスは、AWS IoT とやり取りや AWS クラウドへのログ送信を可能にするコアデバイス IAM ロールを有しています。このデバイスロールは、デフォルトでは S3 バケットへのアクセスを許可しないため、コアデバイスが S3 バケットからコンポーネントアーティファクトを取得できるようにするポリシーを作成して、アタッチする必要があります。

デバイスのロールで S3 バケットへのアクセスが既に許可されている場合は、このステップを省略できます。そうでない場合は、次に示す方法で、アクセスを許可する IAM ポリシーを作成し、ロールにアタッチします。

a. [IAM console] (IAM コンソール) ナビゲーションメニューで、[Policies] (ポリシー) を選択し、[Create policy] (ポリシーの作成) を選択します。

b. JSON タブで、プレースホルダーコンテンツを以下のポリシーに置き換えます。*DOC-EXAMPLE-BUCKET* をコアデバイスでダウンロードするコンポーネントアーティファクトが含まれる S3 バケットの名前に置き換えます。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*"
    }
  ]
}
```

c. [次へ] をクリックします。

d. [ポリシーの詳細セクション] で、[名前] に「**MyGreengrassV2ComponentArtifactPolicy**」と入力します。

e. [ポリシーの作成] を選択します。

f. [IAM console] (IAM コンソール) ナビゲーションメニューで、[Role] (ロール) をクリックし、コアデバイスのロールの名前を選択します。このロール名は、AWS IoT Greengrass Core ソフトウェアをインストールしたときに指定したものです。名前を指定していない場合、デフォルトで `GreengrassV2TokenExchangeRole` が設定されます。

g. [Permissions] (アクセス許可) タブを選択し、[Add permissions] (アクセス許可の追加) を選択してから、[Attach policies] (ポリシーの添付) を選択します。

h. [アクセス許可の追加] ページで、作成した `MyGreengrassV2ComponentArtifactPolicy` ポリシーの横にあるチェックボックスを選択し、[アクセス許可の追加] を選択します。

3. コンポーネント recipe を使用して、AWS IoT Greengrass コンソールにコンポーネントを作成します。

a. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Components] (コンポーネント) 、[Create component] (コンポーネントを作成) の順に選択します。

b. [Component information] (コンポーネント情報) で、[Enter recipe as JSON] (recipe を JSON として入力します) を選択します。プレースホルダ recipe は、次の例のようになります。

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.HelloWorld",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "My first AWS IoT Greengrass component.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "world"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
```

```
        "run": "python3 -u {artifacts:path}/hello_world.py \"{configuration:/
Message}\""
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.HelloWorld/1.0.0/hello_world.py"
        }
      ]
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "run": "py -3 -u {artifacts:path}/hello_world.py \"{configuration:/
Message}\""
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.HelloWorld/1.0.0/hello_world.py"
        }
      ]
    }
  ]
}
```

c. 各 `Artifacts` セクションにあるプレースホルダー URI を、`hello_world.py` オブジェクトの S3 URI に置き換えます。

d. [Create component] (コンポーネントの作成) を選択します。

e. com.example.HelloWorld component ページで、コンポーネントのステータスがデプロイ可能であることを確認します。

AWS IoT Greengrass でコンポーネントを作成する (AWS CLI)

Hello World コンポーネントをアップロードするには

1. AWS アカウント の S3 バケットを使用して、AWS IoT Greengrass コンポーネントアーティファクトをホストします。コンポーネントをコアデバイスにデプロイすると、デバイスがバケットからコンポーネントのアーティファクトをダウンロードします。

既存の S3 バケットを使用するか、次のコマンドを実行してバケットを作成します。このコマンドは、AWS アカウント IDと AWS リージョン を使用してバケットを作成し、一意のバケット名を形成します。*123456789012* を AWS アカウント ID、*region* をこのチュートリアルで使用する AWS リージョン に置き換えます。

```
aws s3 mb s3://greengrass-component-artifacts-123456789012-region
```

以下は、リクエストが成功した場合の出力です。

```
make_bucket: greengrass-component-artifacts-123456789012-region
```

2. コアデバイスが S3 バケット内のコンポーネントアーティファクトにアクセスすることを許可します。

   各コアデバイスは、AWS IoT とやり取りや AWS クラウド へのログ送信を可能にするコアデバイス IAM ロールを有しています。このデバイスロールは、デフォルトでは S3 バケットへのアクセスを許可しないため、コアデバイスが S3 バケットからコンポーネントアーティファクトを取得できるようにするポリシーを作成して、アタッチする必要があります。

   コアデバイスのロールで S3 バケットへのアクセスが既に許可されている場合は、このステップを省略できます。そうでない場合は、次に示す方法で、アクセスを許可する IAM ポリシーを作成し、ロールにアタッチします。

   a. component-artifact-policy.json という名前のファイルを作成して、次の JSON をファイルにコピーします。このポリシーは、S3 バケット内のすべてのファイルへのアクセスを許可します。*DOC-EXAMPLE-BUCKET* を S3 バケットの名前に置き換えます。

      ```
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Action": [
              "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*"
          }
        ]
      }
      ```

b. 次のコマンドを実行して、component-artifact-policy.json のポリシードキュメントからポリシーを作成します。

Linux or Unix

```
aws iam create-policy \\
  --policy-name MyGreengrassV2ComponentArtifactPolicy \\
  --policy-document file://component-artifact-policy.json
```

Windows Command Prompt (CMD)

```
aws iam create-policy ^
  --policy-name MyGreengrassV2ComponentArtifactPolicy ^
  --policy-document file://component-artifact-policy.json
```

PowerShell

```
aws iam create-policy `
  --policy-name MyGreengrassV2ComponentArtifactPolicy `
  --policy-document file://component-artifact-policy.json
```

出力のポリシーメタデータから、ポリシーの Amazon リソースネーム (ARN) をコピーします。この ARN を使用して、次の手順で、このポリシーをコアデバイスのロールにアタッチします。

c. 次のコマンドを実行して、ポリシーをコアデバイスのロールにアタッチします。*GreengrassV2TokenExchangeRole* をコアデバイスのロールの名前に置き換えます。このロール名は、AWS IoT Greengrass Core ソフトウェアをインストールしたときに指定したものです。ポリシー ARN を、前のステップで書き留めた ARN に置き換えます。

Linux or Unix

```
aws iam attach-role-policy \\
  --role-name GreengrassV2TokenExchangeRole \\
  --policy-arn
 arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
```

Windows Command Prompt (CMD)

```
aws iam attach-role-policy ^
  --role-name GreengrassV2TokenExchangeRole ^
  --policy-arn
 arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
```

PowerShell

```
aws iam attach-role-policy `
  --role-name GreengrassV2TokenExchangeRole `
  --policy-arn
 arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
```

成功した場合は、コマンドからの出力はありません。コアデバイスが、この S3 バケットにアップロードしたアーティファクトにアクセスできるようになりました。

3. Hello World Python スクリプトアーティファクトを S3 バケットにアップロードします。

次のコマンドを実行して、AWS IoT Greengrass コア内のスクリプトが存在するバケット内の同じパスにスクリプトをアップロードします。*DOC-EXAMPLE-BUCKET* を S3 バケットの名前に置き換えます。

Linux or Unix

```
aws s3 cp \
  artifacts/com.example.HelloWorld/1.0.0/hello_world.py \
  s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/hello_world.py
```

Windows Command Prompt (CMD)

```
aws s3 cp ^
  artifacts/com.example.HelloWorld/1.0.0/hello_world.py ^
  s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/hello_world.py
```

PowerShell

```
aws s3 cp `
  artifacts/com.example.HelloWorld/1.0.0/hello_world.py `
```

```
  s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/hello_world.py
```

リクエストが成功した場合、コマンドは upload: で始まる行を出力します。

4. アーティファクトの Amazon S3 URI をコンポーネント recipe に追加します。

   Amazon S3 URI は、バケット名とバケット内のアーティファクトオブジェクトへのパスで設定されます。スクリプトアーティファクトの Amazon S3 URI は、前のステップでアーティファクトをアップロードした URI です。この URI は次の例のようになります。*DOC-EXAMPLE-BUCKET* を S3 バケットの名前に置き換えます。

   ```
   s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/hello_world.py
   ```

   アーティファクトを recipe に追加するには、Amazon S3 URI の構造が含まれる `Artifacts` のリストを追加します。

   JSON

   ```
   "Artifacts": [
     {
       "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/
   hello_world.py"
     }
   ]
   ```

   テキストエディタで recipe ファイルを開きます。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

   ```
   nano recipes/com.example.HelloWorld-1.0.0.json
   ```

   アーティファクトを recipe に追加します。recipe ファイルは次の例のようになります。

   ```
   {
     "RecipeFormatVersion": "2020-01-25",
     "ComponentName": "com.example.HelloWorld",
     "ComponentVersion": "1.0.0",
     "ComponentDescription": "My first AWS IoT Greengrass component.",
     "ComponentPublisher": "Amazon",
   ```

```
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "world"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "run": "python3 -u {artifacts:path}/hello_world.py \"{configuration:/
Message}\""
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.HelloWorld/1.0.0/hello_world.py"
        }
      ]
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "run": "py -3 -u {artifacts:path}/hello_world.py \"{configuration:/
Message}\""
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.HelloWorld/1.0.0/hello_world.py"
        }
      ]
    }
  ]
}
```

YAML

```
Artifacts:
```

```
  - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/
hello_world.py
```

テキストエディタで recipe ファイルを開きます。

例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano recipes/com.example.HelloWorld-1.0.0.yaml
```

アーティファクトを recipe に追加します。recipe ファイルは次の例のようになります。

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: My first AWS IoT Greengrass component.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    Message: world
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      run: |
        python3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/
hello_world.py
  - Platform:
      os: windows
    Lifecycle:
      run: |
        py -3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/
hello_world.py
```

5. recipe から、AWS IoT Greengrass コンポーネントを作成します。次のコマンドを実行して、recipe からコンポーネントを作成します。このコンポーネントをバイナリファイルとして提供します。

JSON

```
aws greengrassv2 create-component-version --inline-recipe fileb://recipes/
com.example.HelloWorld-1.0.0.json
```

YAML

```
aws greengrassv2 create-component-version --inline-recipe fileb://recipes/
com.example.HelloWorld-1.0.0.yaml
```

リクエストが成功すると、レスポンスは次の例のようになります。

```
{
  "arn":
 "arn:aws:greengrass:region:123456789012:components:com.example.HelloWorld:versions:1.0.0",
  "componentName": "com.example.HelloWorld",
  "componentVersion": "1.0.0",
  "creationTimestamp": "Mon Nov 30 09:04:05 UTC 2020",
  "status": {
    "componentState": "REQUESTED",
    "message": "NONE",
    "errors": {}
  }
}
```

次のステップでコンポーネントの状態をチェックするために、出力から arn をコピーします。

> **Note**
>
> Hello World コンポーネントは、AWS IoT Greengrass コンソールの [Components] (コンポーネント) ページでも確認できます。

6. コンポーネントが作成され、デプロイする準備が整っていることを確認します。コンポーネントを作成すると、その状態は REQUESTED になります。その後、AWS IoT Greengrass がコンポーネントがデプロイ可能であるかどうかを検証します。次のコマンドを実行して、コンポーネントのステータスを照会し、コンポーネントがデプロイ可能であることを確認します。arn を、前のステップで書き留めた ARN に置き換えます。

```
aws greengrassv2 describe-component --arn
 "arn:aws:greengrass:region:123456789012:components:com.example.HelloWorld:versions:1.0.0"
```

コンポーネントが検証されると、レスポンスでコンポーネントの状態が DEPLOYABLE であることが示されます。

```
{
  "arn":
 "arn:aws:greengrass:region:123456789012:components:com.example.HelloWorld:versions:1.0.0",
  "componentName": "com.example.HelloWorld",
  "componentVersion": "1.0.0",
  "creationTimestamp": "2020-11-30T18:04:05.823Z",
  "publisher": "Amazon",
  "description": "My first Greengrass component.",
  "status": {
    "componentState": "DEPLOYABLE",
    "message": "NONE",
    "errors": {}
  },
  "platforms": [
    {
      "os": "linux",
      "architecture": "all"
    }
  ]
}
```

これで、Hello World コンポーネントが AWS IoT Greengrass で利用できるようになりました。この Greengrass コアデバイスまたは他のコアデバイスにデプロイして戻すことができます。

ステップ 6: コンポーネントをデプロイする

AWS IoT Greengrass を使用して、コンポーネントを個々のデバイスまたはデバイスのグループにデプロイすることができます。コンポーネントをデプロイすると、AWS IoT Greengrass が各ターゲットデバイスにそのコンポーネントのソフトウェアをインストールして実行します。デプロイするコンポーネントと、各コンポーネントにデプロイする設定の更新を指定します。また、デプロイのターゲットとなるデバイスへのデプロイ方法を管理することもできます。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

このセクションでは、Hello World コンポーネントを Greengrass コアデバイスにデプロイして戻します。

コンポーネントをデプロイする (コンソール)

1. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Components] (コンポーネント) を選択します。
2. [Components] (コンポーネント) ページの [My components] (マイコンポーネント) タブで、[com.example.HelloWorld] を選択します。
3. com.example.HelloWorld ページで、[Deploy] (デプロイ) を選択します。
4. [Add to deployment] (デプロイに追加) で、[Create new deployment] (新しいデプロイの作成)、[Next] (次へ) の順に選択します。
5. 「対象を指定する」ページで、次を実行します:
   a. [名前] ボックスに **`Deployment for MyGreengrassCore`** と入力します。
   b. [Deployment target] (デプロイ対象) で、[Core device] (コアデバイス) と、コアデバイスの AWS IoT モノの名前を選択します。このチュートリアルのデフォルト値は です*`MyGreengrassCore`*。
   c. [次へ] をクリックします。
6. [Select components] (コンポーネントを選択) ページの [My components] (マイコンポーネント) 内で、com.example.HelloWorld コンポーネントが選択されていることを確認して、[Next] (次) を選択します。
7. [Configure components] (コンポーネントを設定) ページで、com.example.HelloWorld を選択したら、次の操作を行います。
   a. [Configure component] (コンポーネントを設定) を選択します。
   b. [Configuration update] (設定を更新) の [Configuration to merge] (マージの設定) で、次の設定を入力します。

      ```
      {
        "Message": "universe"
      }
      ```

      この設定更新は、このデプロイ内にあるデバイスの Hello World `Message` パラメータを `universe` に設定します。
   c. [確認] を選択します。

d. [次へ] をクリックします。

8. [Configure advanced settings] (詳細設定) ページはデフォルト設定のままにし、[Next] (次へ) を選択します。
9. [Review] ページで、[デプロイ] を選択します。
10. デプロイが正常に完了すていることを確認します。デプロイには数分かかる場合があります。Hello World のログをチェックして、変更を確認します。Greengrass コアデバイスで、次のコマンドを実行します。

    Linux or Unix

    ```
    sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
    ```

    Windows Command Prompt (CMD)

    ```
    type C:\greengrass\v2\\logs\\com.example.HelloWorld.log
    ```

    PowerShell

    ```
    gc C:\greengrass\v2\\logs\\com.example.HelloWorld.log -Tail 10 -Wait
    ```

    次の例に示すようなメッセージが表示されます。

    ```
    Hello, universe! Greetings from your first Greengrass component.
    ```

    > **Note**
    >
    > ログメッセージが変更されない場合、デプロイが失敗しているか、コアデバイスに到達していません。これは、コアデバイスがインターネットに接続されていない、あるいは S3 バケットからアーティファクトを取得するために適切な権限がない場合に発生します。コアデバイスで次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアのログファイルを確認します。このファイルは、Greengrass コアデバイスのデプロイサービスからのログが含まれます。
    >
    > Linux or Unix
    >
    > ```
    > sudo tail -f /greengrass/v2/logs/greengrass.log
    > ```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\\logs\\greengrass.log
```

type コマンドは、ファイルのコンテンツを端末に書き込みます。このコマンドを複数回実行して、ファイル内の変更を確認してください。

PowerShell

```
gc C:\greengrass\v2\\logs\\greengrass.log -Tail 10 -Wait
```

詳細については、「トラブルシューティング AWS IoT Greengrass V2」を参照してください。

コンポーネントをデプロイする (AWS CLI)

Hello World コンポーネントをデプロイするには

1. 開発コンピュータ上で、hello-world-deployment.json という名前のファイルを作成して、次の JSON をファイルにコピーします。このファイルは、デプロイするコンポーネントと設定を定義しています。

```
{
  "components": {
    "com.example.HelloWorld": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "merge": "{\"Message\":\"universe\"}"
      }
    }
  }
}
```

この設定ファイルは、前の手順で開発してパブリッシュした Hello World コンポーネントのバージョン 1.0.0 をデプロイするように指定します。configurationUpdate は、JSON エンコードされた文字列でコンポーネント設定をマージすることを指定しています。この設定更新

は、このデプロイ内にあるデバイスの Hello World `Message` パラメータを `universe` に設定します。

2. 次のコマンドを実行して、Greengrass コアデバイスにコンポーネントをデプロイします。個々のデバイスである「モノ」、またはデバイスのグループである「モノのグループ」にデプロイすることができます。をコアデバイスのAWS IoTモノの名前*MyGreengrassCore*に置き換えます。

   Linux or Unix

   ```
   aws greengrassv2 create-deployment \
     --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" \
     --cli-input-json file://hello-world-deployment.json
   ```

   Windows Command Prompt (CMD)

   ```
   aws greengrassv2 create-deployment ^
     --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" ^
     --cli-input-json file://hello-world-deployment.json
   ```

   PowerShell

   ```
   aws greengrassv2 create-deployment `
     --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" `
     --cli-input-json file://hello-world-deployment.json
   ```

   コマンドは次の出力例のようなレスポンスを返します。

   ```
   {
     "deploymentId": "deb69c37-314a-4369-a6a1-3dff9fce73a9",
     "iotJobId": "b5d92151-6348-4941-8603-bdbfb3e02b75",
     "iotJobArn": "arn:aws:iot:region:account-id:job/b5d92151-6348-4941-8603-
   bdbfb3e02b75"
   }
   ```

3. デプロイが正常に完了すていることを確認します。デプロイには数分かかる場合があります。Hello World のログをチェックして、変更を確認します。Greengrass コアデバイスで、次のコマンドを実行します。

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\\logs\\com.example.HelloWorld.log
```

PowerShell

```
gc C:\greengrass\v2\\logs\\com.example.HelloWorld.log -Tail 10 -Wait
```

次の例に示すようなメッセージが表示されます。

```
Hello, universe! Greetings from your first Greengrass component.
```

> **Note**
>
> ログメッセージが変更されない場合、デプロイが失敗しているか、コアデバイスに到達していません。これは、コアデバイスがインターネットに接続されていない、あるいは S3 バケットからアーティファクトを取得するために適切な権限がない場合に発生します。コアデバイスで次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアのログファイルを確認します。このファイルは、Greengrass コアデバイスのデプロイサービスからのログが含まれます。
>
> Linux or Unix
>
> ```
> sudo tail -f /greengrass/v2/logs/greengrass.log
> ```
>
> Windows Command Prompt (CMD)
>
> ```
> type C:\greengrass\v2\\logs\\greengrass.log
> ```
>
> type コマンドは、ファイルのコンテンツを端末に書き込みます。このコマンドを複数回実行して、ファイル内の変更を確認してください。

PowerShell

```
gc C:\greengrass\v2\\logs\\greengrass.log -Tail 10 -Wait
```

詳細については、「トラブルシューティング AWS IoT Greengrass V2」を参照してください。

次のステップ

これで、このチュートリアルは終了です。AWS IoT Greengrass Core ソフトウェアと Hello World コンポーネントがデバイス上で実行されています。また、Hello World コンポーネントは AWS IoT Greengrass クラウドサービスで利用できる状態で、他のデバイスにデプロイできます。このチュートリアルで説明しているトピックの詳細については、以下を参照してください。

- AWS IoT Greengrass コンポーネントの作成
- コアデバイスにデプロイするコンポーネントをパブリッシュ
- デバイスに AWS IoT Greengrass コンポーネントのデプロイ

AWS IoT Greengrass コアデバイスをセットアップする

このセクションのタスクを完了し、AWS IoT Greengrass Core ソフトウェアをインストール、設定、および実行します。

> **Note**
>
> ここでは、AWS IoT Greengrass Core ソフトウェアの高度なインストールと設定について説明します。AWS IoT Greengrass V2 を初めて使用する場合は、最初に [getting started tutorial] (入門チュートリアル) を完了してコアデバイスをセットアップし、AWS IoT Greengrass の機能を確認することをお勧めします。

サポートされているプラットフォームと要件

開始する前に、AWS IoT Greengrass Core ソフトウェアをインストールして実行するには、以下の要件を満たしていることを確認してください。

> **Tip**
>
> [AWS Partner Device Catalog] (パートナーデバイスカタログ) で AWS IoT Greengrass V2 に該当するデバイスを検索できます。

トピック

サポートされているプラットフォーム

AWS IoT Greengrass は、次のプラットフォームを実行するデバイスを正式にサポートしています。このリストに含まれていないプラットフォームを持つデバイスは動作する可能性がありますが、AWS IoT Greengrass はこれらの指定されたプラットフォームでのみテストします。

Linux

アーキテクチャ:

- Armv7l
- Armv8 (AArch64)
- x86_64

Windows

アーキテクチャ:

- x86_64

バージョン:

- Windows 10
- Windows 11
- [Windows Server 2019]
- Windows Server 2022

> **Note**
>
> 一部の AWS IoT Greengrass 機能は、現在 Windows デバイスではサポートされていません。詳細については、「オペレーティングシステム別 Greengrass 機能の互換性」 および 「Windows デバイスの機能に関する考慮事項」 を参照してください。

Linux プラットフォームは、Docker コンテナで AWS IoT Greengrass V2 を実行することもできます。詳細については、「Docker コンテナで AWS IoT Greengrass Core ソフトウェアを実行する」を参照してください。

カスタム Linux ベースのオペレーティングシステムを構築するには、`meta-aws`プロジェクト AWS IoT Greengrass V2で の BitBake レシピを使用できます。この`meta-aws`プロジェクトでは、OpenEmbeddedおよび Yocto Project ビルドフレームワークで構築された組み込み Linux システムにAWSエッジソフトウェア機能を構築するために使用できるレシピを提供します。Yocto Project は、ハードウェアのアーキテクチャに関係なく、組み込みアプリケーション用のカスタム Linux ベー

スのシステムをビルドするのに役立つ、オープンソースのコラボレーションプロジェクトです。の BitBake レシピは、デバイスで AWS IoT Greengrass Core ソフトウェアAWS IoT Greengrass V2をインストール、設定、および自動的に実行します。

デバイスの要件

AWS IoT Greengrass Core ソフトウェア v2.x をインストールして実行するには、デバイスが次の要件を満たしている必要があります。

> **Note**
>
> AWS IoT Greengrass の AWS IoT Device Tester を使用すると、デバイスが AWS IoT Greengrass Core ソフトウェアを実行し、AWS クラウド と通信できることを確認できます。詳細については、「AWS IoT Device Tester for AWS IoT Greengrass V2 を使用する」を参照してください。

Linux

- AWS IoT Greengrass V2 をサポートする AWS リージョン を使用します。サポートされているリージョンのリストについては、「AWS 全般のリファレンス」の「AWS IoT Greengrass V2 の エンドポイントとクォータ」を参照してください。
- AWS IoT Greengrass Core ソフトウェアに対して利用可能な最低 256 MB のディスク空き容量。この要件には、コアデバイスにデプロイされたコンポーネントは含まれません。
- AWS IoT Greengrass Core ソフトウェアに割り当てられる最小 96 MB RAM。この要件には、コアデバイスで実行されるコンポーネントは含まれません。詳細については、「JVM オプションでメモリ割り当てを制御する」を参照してください。
- Java ランタイム環境 (JRE) バージョン 8 以降。デバイスの PATH 環境変数で Java が利用可能になっている必要があります。Java を使用してカスタムコンポーネントを開発するには、Java Development Kit (JDK) をインストールする必要があります。Amazon Corretto または OpenJDK の長期サポートバージョンを使用することをお勧めします。バージョン 8 以降が必要です。
- [GNU C Library] (GNU C ライブラリ)(glibc) バージョン 2.25 以降。
- AWS IoT Greengrass Core ソフトウェアは、ルートユーザーとして実行する必要があります。例えば `sudo` を使用します。
- `root` などの AWS IoT Greengrass Core ソフトウェアを実行するルートユーザーは、任意のユーザーおよび任意のグループで `sudo` を実行する権限があることが必要です。`/etc/`

sudoers ファイルは、このユーザーに sudo を他のグループとして実行する権限が与えられていることが必要です。/etc/sudoers のユーザーの権限は、次の例のようになります。

```
root    ALL=(ALL:ALL) ALL
```

- コアデバイスは、一連のエンドポイントおよびポートへのアウトバウンド要求を実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。
- /tmp ディレクトリは exec 権限でマウントする必要があります。
- 次のすべてのシェルコマンド:
  - ps -ax -o pid,ppid
  - sudo
  - sh
  - kill
  - cp
  - chmod
  - rm
  - ln
  - echo
  - exit
  - id
  - uname
  - grep
- デバイスでは、次のオプションのシェルコマンドが必要な場合もあります。
  - (オプション) systemctl。このコマンドは、AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定するために使用します。
  - (オプション) useradd、groupadd、および usermod。これらのコマンドは、ggc_user システムユーザーと ggc_group システムグループを設定するために使用します。
  - (オプション) mkfifo。このコマンドは、Lambda 関数をコンポーネントとして実行するために使用します。
- コンポーネントプロセスのシステムリソース制限を設定するには、デバイスで Linux カーネルバージョン 2.6.24 以降を実行する必要があります。

- Lambda 関数を実行するには、デバイスが追加の要件を満たしている必要があります。詳細については、「Lambda 関数の要件」を参照してください。

Windows

- AWS IoT Greengrass V2 をサポートする AWS リージョン を使用します。サポートされているリージョンのリストについては、「AWS 全般のリファレンス」の「AWS IoT Greengrass V2 の エンドポイントとクォータ」を参照してください。
- AWS IoT Greengrass Core ソフトウェアに対して利用可能な最低 256 MB のディスク空き容量。この要件には、コアデバイスにデプロイされたコンポーネントは含まれません。
- AWS IoT Greengrass Core ソフトウェアに割り当てられる最小 160 MB RAM。この要件には、コアデバイスで実行されるコンポーネントは含まれません。詳細については、「JVM オプションでメモリ割り当てを制御する」を参照してください。
- Java ランタイム環境 (JRE) バージョン 8 以降。デバイスの PATH システム変数で Java が利用可能になっている必要があります。Java を使用してカスタムコンポーネントを開発するには、Java Development Kit (JDK) をインストールする必要があります。Amazon Corretto または OpenJDK の長期サポートバージョンを使用することをお勧めします。バージョン 8 以降が必要です。

  > **Note**
  >
  > Greengrass nucleus のバージョン 2.5.0 を使用するには、64 ビット版の Java Runtime Environment (JRE) を使用する必要があります。Greengrass nucleus バージョン 2.5.1 は 32 ビットおよび 64 ビットの JRE をサポートします。

- AWS IoT Greengrass Coreソフトウェアをインストールするユーザーは、管理者である必要があります。
- AWS IoT Greengrass Core ソフトウェアをシステムサービスとしてインストールする必要があります。ソフトウェアをインストールするとき、`--setup-system-service true` を指定します。
- コンポーネントプロセスを実行する各ユーザーは LocalSystem アカウントに存在し、ユーザーの名前とパスワードは LocalSystem アカウントの認証情報マネージャーインスタンスにある必要があります。AWS IoT Greengrass Core ソフトウェアをインストールする手順に従うと、このユーザーを設定できます。

- コアデバイスは、一連のエンドポイントおよびポートへのアウトバウンド要求を実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

Lambda 関数の要件

Lambda 関数を実行するには、デバイスが次の要件を満たしている必要があります。

- Linux ベースのオペレーティングシステム。
- デバイスには `mkfifo` シェルコマンドが必要です。
- デバイスでは、Lambda 関数に必要なプログラミング言語ライブラリを実行する必要があります。必須ライブラリを、デバイスにインストールし、PATH 環境変数に追加する必要があります。Greengrass は、Lambda でサポートされるすべてのバージョンの Python、Node.js、Java ランタイムをサポートします。Greengrass は、非推奨となった Lambda ランタイムバージョンに追加の制限を適用しません。Lambda ランタイムの AWS IoT Greengrass サポートの詳細については、「AWS Lambda 関数を実行する」を参照してください。
- コンテナ化された Lambda 関数を実行するには、デバイスが次の要件を満たしている必要があります。
  - Linux kernel バージョン 4.4 以降。
  - カーネルは cgroups v1 をサポートしている必要があり、次の cgroups を有効にしてマウントする必要があります。
    - - コンテナ化された Lambda 関数のメモリ制限を設定するための AWS IoT Greengrass のメモリ cgroup。
    - - システムデバイスまたはボリュームにアクセスするコンテナ化された Lambda 関数のためのデバイスcgroup。

    AWS IoT Greengrass Core ソフトウェアは、cgroups v2 をサポートしていません。

    この要件を満たすには、次の Linux カーネルパラメータを使用してデバイスを起動します。

    ```
    cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
    ```

> **Tip**
>
> Raspberry Pi で、`/boot/cmdline.txt` ファイルを編集して、デバイスのカーネルパラメータを設定します。

- デバイスで次の Linux カーネル設定を有効にする必要があります。
  - 名前空間:
    - `CONFIG_IPC_NS`
    - `CONFIG_UTS_NS`
    - `CONFIG_USER_NS`
    - `CONFIG_PID_NS`
  - Cgroups:
    - `CONFIG_CGROUP_DEVICE`
    - `CONFIG_CGROUPS`
    - `CONFIG_MEMCG`
  - Others:
    - `CONFIG_POSIX_MQUEUE`
    - `CONFIG_OVERLAY_FS`
    - `CONFIG_HAVE_ARCH_SECCOMP_FILTER`
    - `CONFIG_SECCOMP_FILTER`
    - `CONFIG_KEYS`
    - `CONFIG_SECCOMP`
    - `CONFIG_SHMEM`

> **Tip**
>
> Linux カーネルパラメータを検証して設定する方法については、Linux ディストリビューションのドキュメントを確認してください。また、AWS IoT Device Tester を使用して、デバイスがこれらの要件を満たしていることを AWS IoT Greengrass に確認させることもできます。詳細については、「AWS IoT Device Tester for AWS IoT Greengrass V2 を使用する」を参照してください。

Windows デバイスの機能に関する考慮事項

一部の AWS IoT Greengrass 機能は、現在 Windows デバイスではサポートされていません。Windows デバイスが要件を満たしているかどうかを確認するには、機能の違いを確認してください。詳細については、「オペレーティングシステム別 Greengrass 機能の互換性」を参照してください。

AWS アカウント のセットアップ

AWS アカウント がない場合は、以下のステップを実行して作成します。

AWS アカウント にサインアップするには

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、通話呼び出しを受け取り、電話のキーパッドを使用して検証コードを入力するように求められます。

   AWS アカウントにサインアップすると、AWS アカウントのルートユーザーが作成されます。ルートユーザーには、アカウントのすべての AWS のサービスとリソースへのアクセス権があります。セキュリティのベストプラクティスとして、管理ユーザーに管理アクセスを割り当て、ルートユーザーアクセスが必要なタスクを実行する場合にのみ、ルートユーザーを使用してください。

管理者ユーザーを作成するには、以下のいずれかのオプションを選択します。

管理者を管理する方法を 1 つ選択します	目的	方法	以下の操作も可能
IAM Identity	短期の認証情報を使用して AWS にアクセスします。	AWS IAM Identity Center ユーザーガイドの「開始方法」の手順に従います。	AWS Command Line Interface ユーザーガイドの「AWS IAM Identity Center を使用する

管理者を管理する方法を 1 つ選択します	目的	方法	以下の操作も可能
Center 内 (推奨)	これはセキュリティのベストプラクティスと一致しています。ベストプラクティスの詳細については、IAM ユーザーガイドの「IAM でのセキュリティのベストプラクティス」を参照してください。		ための AWS CLI の設定」に従って、プログラムによるアクセスを設定します。
IAM 内 (非推奨)	長期認証情報を使用して AWS にアクセスする。	IAM ユーザーガイドの「最初の IAM 管理者のユーザーおよびグループの作成」の手順に従います。	IAM ユーザーガイドの「IAM ユーザーのアクセスキーの管理」に従って、プログラムによるアクセスを設定します。

AWS IoT Greengrass Core ソフトウェアをインストールします。

AWS IoT Greengrass は AWS をエッジデバイスに拡張します。これによりエッジデバイスは生成するデータに対してアクションを実行しながら、管理、分析、耐久性のあるストレージのために AWS クラウド を使用できます。AWS IoT Greengrass Core ソフトウェアをエッジデバイスにインストールして、AWS IoT Greengrass と AWS クラウド を統合します。

> **Important**
>
> AWS IoT Greengrass Core ソフトウェアをダウンロードしてインストールする前に、お使いのコアデバイスが AWS IoT Greengrass Core ソフトウェア v2.0 をインストールして実行するための要件を満たしていることを確認してください。

AWS IoT Greengrass Core ソフトウェアには、デバイスを Greengrass コアデバイスとして設定するインストーラが含まれています。インストーラを実行する際は、ルートフォルダや使用する AWS リージョン などのオプションを設定できます。必要な AWS IoT および IAM リソース作成をインストーラに作成させることも選択できます。また、ローカル開発ツールをデプロイして、カスタムコンポーネント開発に使用するデバイスを設定することもできます。

AWS IoT Greengrass Core ソフトウェアは、AWS クラウド への接続および操作を行なうためには、以下の AWS IoT および IAM リソースを必要とします。

- AWS IoT のモノ デバイスを AWS IoT モノとして登録するとき、そのデバイスはデジタル証明書を使用して AWS で認証できます。この証明書は、デバイスが AWS IoT と AWS IoT Greengrass と通信できるようにします。詳細については、「AWS IoT Greengrass のデバイス認証と認可」を参照してください。
- (オプション) AWS IoT モノグループ。モノグループを使用して Greengrass コアデバイスのフリートを管理します。ソフトウェアコンポーネントをデバイスにデプロイするとき、個々のデバイスまたはデバイスのグループのどちらにデプロイするのかを選択することができます。デバイスをモノグループに追加すると、そのモノグループのソフトウェアコンポーネントをデバイスにデプロイできます。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。
- IAM ロール。Greengrass コアデバイスは、AWS IoT Core 認証情報プロバイダを使用して AWS サービスへの呼び出しを認証する際に、IAM ロールを使用します。このロールにより、デバイスは とやり取りしAWS IoT、Amazon CloudWatch Logs にログを送信し、Amazon Simple Storage Service (Amazon S3) からカスタムコンポーネントアーティファクトをダウンロードできるようになります。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。
- AWS IoT ロールエイリアス。Greengrass コアデバイスは、使用する IAM ロールの識別にロールエイリアスを使用します。ロールエイリアスを使用すると、IAM ロールを変更してもデバイス設定は同じ状態を保つことができます。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS サービスへの直接呼び出しを認証する」を参照してください。

次のいずれかのオプションを選択して、デバイスに AWS IoT Greengrass Core ソフトウェアをインストールします。

- クイックインストール

  Greengrass コアデバイスをできるだけ少ないステップでセットアップするには、このオプションを選択します。必要な AWS IoT および IAM リソースはインストーラが作成します。このオプショ

ンでは、AWS アカウント でリソースを作成するために、インストーラに AWS 認証情報を提供する必要があります。

このオプションを使用して、ファイアウォールまたはネットワークプロキシの向こう側にインストールすることはできません。デバイスがファイアウォールまたはネットワークプロキシの向こう側にある場合は、手動インストールを検討してください。

詳細については、「自動リソースプロビジョニング機能を備えた AWS IoT Greengrass Core ソフトウェアをインストール」を参照してください。

- 手動インストール

  必要となる AWS リソースを手動で作成する場合、またはファイアウォールもしくはネットワークプロキシの向こう側にインストールを行う場合は、このオプションを選択します。手動でインストールすることで、インストーラに AWS アカウント でリソースを作成する権限を与える必要がなくなります。必要となる AWS IoT と IAM リソースはユーザーが作成するためです。また、ポート 443 で、またはネットワークプロキシ経由で接続するようにデバイスを設定することもできます。また、ハードウェアセキュリティモジュール (HSM)、トラステッドプラットフォームモジュール (TPM)、またはその他の暗号化要素にプライベートキーおよび証明書を保存して使用するように、AWS IoT Greengrass Core ソフトウェアを設定することもできます。

  詳細については、「手動リソースプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。

- AWS IoT フリートプロビジョニングを使用したインストール

  AWS IoT フリートプロビジョニングテンプレートから必要な AWS リソースを作成するには、このオプションを選択します。このオプションは、フリート内で類似したデバイスを作成する場合や、車両やスマートホームデバイスなど、顧客が後でアクティブ化するデバイスを製造する場合に選択します。デバイスはクレーム証明書を使用して AWS リソースの認証とプロビジョニングを行います。これには、デバイスが通常のオペレーション用に AWS クラウド に接続するために使用する X.509 クライアント証明書などがあります。クレーム証明書は、製造時にデバイスのハードウェアに埋め込みまたはフラッシュできます。また同じクレーム証明書とキーを使用して、複数のデバイスをプロビジョニングできます。また、ポート 443 で、またはネットワークプロキシ経由で接続するようにデバイスを設定することもできます。

  詳細については、「AWS IoT フリートプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。

- カスタムプロビジョニングを使用したインストール

必要な AWS リソースをプロビジョニングするカスタム Java アプリケーションを作成するには、このオプションを選択します。独自の X.509 クライアント証明書を作成する場合、またはプロビジョニングプロセスのより細かい制御が必要な場合は、このオプションを選択できます。AWS IoT Greengrass は、カスタムプロビジョニングアプリケーションと AWS IoT Greengrass Core ソフトウェアインストーラとの間の情報交換のために実装できるインターフェイスを提供します。

詳細については、「カスタムリソースプロビジョニングを使用して AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。

AWS IoT Greengrass は、AWS IoT Greengrass Core ソフトウェアを実行するコンテナ化された環境も提供します。Docker コンテナで AWS IoT Greengrass を実行するには、Dockerfile を使用します。

トピック

- 自動リソースプロビジョニング機能を備えた AWS IoT Greengrass Core ソフトウェアをインストール
- 手動リソースプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする
- AWS IoT フリートプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする
- カスタムリソースプロビジョニングを使用して AWS IoT Greengrass Core ソフトウェアをインストールする
- インストーラ引数

自動リソースプロビジョニング機能を備えた AWS IoT Greengrass Core ソフトウェアをインストール

AWS IoT Greengrass Core ソフトウェアには、デバイスを Greengrass コアデバイスとして設定するインストーラが含まれています。デバイスをすばやくセットアップするために、インストーラは AWS IoT モノ、AWS IoT モノグループ、IAM ロール、コアデバイスの動作に必要な AWS IoT ロールエイリアスをプロビジョニングできます。インストーラは、ローカル開発ツールをコアデバイスにデプロイすることもできるため、デバイスを使用してカスタムソフトウェアコンポーネントを開発とテストできます。インストーラは、これらのリソースをプロビジョニングしてデプロイを作成すには、AWS 認証情報が必要です。

デバイスに AWS 認証情報を提供できない場合、コアデバイスの動作に必要な AWS リソースをプロビジョニングできます。開発ツールをコアデバイスにデプロイして、開発デバイスとして使用することもできます。これにより、インストーラの実行時にデバイスに与える許可を減らすことができます。詳細については、「手動リソースプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。

> **Important**
>
> AWS IoT Greengrass Core ソフトウェアをダウンロードする前に、コアデバイスが AWS IoT Greengrass Core ソフトウェア v2.0 をインストールして実行するための要件を満たしていることを確認してください。

トピック

- デバイス環境をセットアップする
- デバイスに AWS 認証情報の提供
- AWS IoT Greengrass Core ソフトウェアをダウンロードする
- AWS IoT Greengrass Core ソフトウェアをインストールします。

デバイス環境をセットアップする

このセクションのステップに従って、AWS IoT Greengrass コアデバイスとして使用する Linux または Windows デバイスをセットアップします。

Linux デバイスをセットアップする

AWS IoT Greengrass V2 の Linux デバイスをセットアップするには

1. AWS IoT Greengrass Core ソフトウェアを実行するために必要な Java ランタイムをインストールします。Amazon Corretto または OpenJDK の長期サポートバージョンを使用することをお勧めします。バージョン 8 以降が必要です。次のコマンドは、デバイスに OpenJDK をインストールする方法を示しています。

   - Debian ベースまたは Ubuntu ベースのディストリビューションの場合:

     ```
     sudo apt install default-jdk
     ```

   - Red Hat ベースのディストリビューションの場合 :

```
sudo yum install java-11-openjdk-devel
```

- 複数 Amazon Linux 2:

```
sudo amazon-linux-extras install java-openjdk11
```

- 複数 Amazon Linux 2023:

```
sudo dnf install java-11-amazon-corretto -y
```

インストールが完了したら、次のコマンドを実行して Java が Linux デバイスで実行されていることを確認します。

```
java -version
```

このコマンドは、デバイス上で実行されている Java のバージョンを出力します。例えば、Debian ベースのディストリビューションでは、出力は次のサンプルのようになります。

```
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

2. (オプション) デバイスにコンポーネントを実行するデフォルトのシステムユーザーおよびグループを作成します。AWS IoT Greengrass Core ソフトウェアインストーラは、`--component-default-user` インストーラ引数でこのユーザーとグループを作成させるという選択もあります。詳細については、「インストーラ引数」を参照してください。

```
sudo useradd --system --create-home ggc_user
sudo groupadd --system ggc_group
```

3. AWS IoT Greengrass Core ソフトウェア (通常は `root`) を実行するユーザーが、sudo を任意のユーザーと任意のグループに実行する許可があることを確認してください。

   a. `/etc/sudoers` ファイルを開くには、次のコマンドを実行します。

   ```
   sudo visudo
   ```

   b. ユーザーの権限が次の例のようになっていることを確認します。

```
root    ALL=(ALL:ALL) ALL
```

4. (オプション) コンテナ化された Lambda 関数を実行するには、cgroups v1 を有効にし、メモリとデバイスの cgroups を有効にしてマウントする必要があります。コンテナ化された Lambda 関数を実行する予定がない場合、この手順を省略できます。

   これらの cgroups オプションを有効にするには、次の Linux カーネルパラメータを使用してデバイスを起動します。

   ```
   cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
   ```

   デバイスのカーネルパラメータを確認および設定するための情報については、オペレーティングシステムおよびブートローダーのドキュメントを参照してください。指示に従って、カーネルパラメータを永続的に設定します。
5. デバイスの要件 にある要件リストで示されているように、その他の必要となる依存関係をすべてデバイスにインストールします。

Windows デバイスをセットアップする

> **Note**
>
> この機能は、Greengrass nucleus コンポーネントの v2.5.0 以降に利用できます。

AWS IoT Greengrass V2 の Windows デバイスをセットアップするには

1. AWS IoT Greengrass Core ソフトウェアを実行するために必要な Java ランタイムをインストールします。Amazon Corretto または OpenJDK の長期サポートバージョンを使用することをお勧めします。バージョン 8 以降が必要です。
2. PATH システム変数で Java が使用可能か確認し、そうでない場合は追加します。 LocalSystem アカウントは AWS IoT Greengrass Core ソフトウェアを実行するため、ユーザーの PATH ユーザー変数ではなく、PATH システム変数に Java を追加する必要があります。以下の操作を実行します。

   a. Windows キーを押してスタートメニューを開きます。
   b. **environment variables** を入力して、スタートメニューからシステムオプションを検索します。

c. スタートメニューの検索結果から [Edit the system environment variables] (システム環境変数を編集) をクリックして、[System properties] (システムプロパティ) ウィンドウを開きます。

d. [Environment variables...] (環境変数...) を選択して、[Environment Variables] (環境可変) ウィンドウを開きます。

e. [System variables] (システム変数) で、[Path] (パス) 、[Edit] (編集) の順に選択します。[Edit environment variable] (環境変数の編集) ウィンドウでは、個別の行に各パスを表示できます。

f. Java インストールの `bin` フォルダへのパスが存在しているかを確認します。このパスは、次の例のように表示されます。

```
C:\\Program Files\\Amazon Corretto\\jdk11.0.13_8\\bin
```

g. [Path] (パス) で Java インストールの `bin` フォルダが で見つからない場合は、[New] (新規) を選択してこれを追加した上で、[OK] を選択します。

3. 管理者として Windows コマンドプロンプト `cmd.exe` を開きます。

4. Windows デバイスの LocalSystem アカウントにデフォルトユーザーを作成します。*#####*を安全なパスワードに置き換えます。

```
net user /add ggc_user password
```

> **Tip**
>
> Windows の構成によっては、ユーザーのパスワードの期限切れが、将来の日付に設定されている場合があります。Greengrass アプリケーションの動作を継続させるためには、パスワードの有効期限を追跡し、その期限が切れる前に更新します。ユーザーのパスワードには、期限切れを起こさないような設定も可能です。
>
> - ユーザーとパスワードの有効期限を確認するには、次のコマンドを実行します。
>
>   ```
>   net user ggc_user | findstr /C:expires
>   ```
>
> - ユーザーのパスワードが期限切れにならないように設定するには、次のコマンドを実行します。
>
>   ```
>   wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
>   ```

- wmic コマンドが廃止された Windows 10 以降を使用している場合は、次の PowerShell コマンドを実行します。

```
Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
 'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
```

5. デバイスに Microsoft から PsExec ユーティリティをダウンロードしてインストールします。
6. PsExec ユーティリティを使用して、デフォルトユーザーのユーザー名とパスワードを LocalSystem アカウントの認証情報マネージャーインスタンスに保存します。*#####*を以前に設定したユーザーのパスワードに置き換えます。

```
psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
```

PsExec License Agreement が開いたら、Accept を選択し、ライセンスに同意してコマンドを実行します。

> **Note**
>
> Windows デバイスでは、 LocalSystem アカウントが Greengrass nucleus を実行するため、 PsExec ユーティリティを使用してデフォルトのユーザー情報を LocalSystem アカウントに格納する必要があります。認証情報マネージャーアプリケーションを使用すると、この情報は アカウントではなく、現在ログオンしているユーザーの Windows LocalSystem アカウントに保存されます。

デバイスに AWS 認証情報の提供

デバイスに AWS 認証情報を提供して、インストーラが必要な AWS リソースをプロビジョニングできるようにします。必要な許可の詳細については、「インストーラがリソースをプロビジョニングするための最小限の IAM ポリシー」を参照してください。

デバイスに AWS 認証情報を提供するには

- デバイスに AWS 認証情報を提供して、インストーラがコアデバイスの AWS IoT と IAM リソースをプロビジョニングできるようにします。セキュリティを強化するには、プロビジョニングに必要な最小限の許可のみを与える IAM ロールの一時的な認証情報を取得することをお勧めしま

す。詳細については、「インストーラがリソースをプロビジョニングするための最小限の IAM ポリシー」を参照してください。

> **Note**
>
> インストーラが認証情報を保存したり保管することはありません。

デバイスに、次のいずれかを実行して、認証情報を取得して AWS IoT Greengrass Core ソフトウェアのインストーラを利用できるようにしてください。

- （推奨) からの一時認証情報を使用する AWS IAM Identity Center

  a. IAM Identity Center からアクセスキー ID、シークレットアクセスキー、およびセッショントークンを指定します。詳細については、IAM Identity Center ユーザーガイドの「一時的な認証情報の取得と更新」の「認証情報の手動更新」を参照してください。

  b. 次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアに認証情報を提供します。

  Linux or Unix

  ```
  export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
  export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
  export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
  ```

  Windows Command Prompt (CMD)

  ```
  set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
  set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
  set AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
  ```

  PowerShell

  ```
  $env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
  $env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
  $env:AWS_SESSION_TOKEN="AQoDYXdzEJr1K...o5OytwEXAMPLE="
  ```

- IAM ロールの一時的なセキュリティ認証情報を使用します。

a. 継承する IAM ロールから、アクセスキー ID、シークレットアクセスキー、セッショントークンを提供します。これらの認証情報を取得する方法の詳細については、「IAM ユーザーガイド」の「一時的なセキュリティ認証情報のリクエスト」を参照してください。

b. 次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアに認証情報を提供します。

Linux or Unix

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

Windows Command Prompt (CMD)

```
set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
set AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

PowerShell

```
$env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
$env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
$env:AWS_SESSION_TOKEN="AQoDYXdzEJr1K...o5OytwEXAMPLE="
```

- IAM ユーザーからの長期的な認証情報を使用する:

a. IAM ユーザーのアクセスキー ID とシークレットアクセスキーを提供します。後で削除するプロビジョニング用の IAM ユーザーを作成できます。ユーザーに付与する IAM ポリシーについては、「」を参照してくださいインストーラがリソースをプロビジョニングするための最小限の IAM ポリシー。長期認証情報を取得する方法の詳細については、「IAM ユーザーガイド」の「IAM ユーザーのアクセスキー管理」を参照してください。

b. 次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアに認証情報を提供します。

Linux or Unix

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

Windows Command Prompt (CMD)

```
set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

PowerShell

```
$env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
$env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
```

c. （オプション) Greengrass デバイスをプロビジョニングする IAM ユーザーを作成した場合は、そのユーザーを削除します。

d. （オプション) 既存の IAM ユーザーのアクセスキー ID とシークレットアクセスキーを使用した場合は、そのユーザーのキーを更新して無効になるようにします。詳細については、「 ユーザーガイド」の「アクセスキーの更新」を参照してください。 AWS Identity and Access Management

AWS IoT Greengrass Core ソフトウェアをダウンロードする

AWS IoT Greengrass Core ソフトウェアの最新バージョンを次の場所からダウンロードできます。

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアの特定のバージョンは、次の場所からダウンロードできます。*#####*をダウンロードするバージョンに置き換えます。
>
> ```
> https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip
> ```

AWS IoT Greengrass Core ソフトウェアをダウンロードするには

1. コアデバイス上で、AWS IoT Greengrass Core ソフトウェアを `greengrass-nucleus-latest.zip` という名前のファイルにダウンロードします。

   Linux or Unix

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip > greengrass-nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip > greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip -OutFile greengrass-nucleus-latest.zip
   ```

   このソフトウェアをダウンロードすると、Greengrass Core ソフトウェアのライセンス契約に同意したものと見なされます。

2. (オプション) Greengrass nucleus ソフトウェア署名を確認するには

   > **Note**
   >
   > この機能は、Greengrass nucleus バージョン 2.9.5 以降で使用できます。

   a. 以下のコマンドを使用して、Greengrass nucleus アーティファクトの署名を確認します。

      Linux or Unix

      ```
      jarsigner -verify -certs -verbose greengrass-nucleus-latest.zip
      ```

Windows Command Prompt (CMD)

インストールする JDK のバージョンによって、ファイル名が異なる場合があります。インストールした JDK のバージョンに *jdk17.0.6_10* を置き換えてください。

```
"C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe" -
verify -certs -verbose greengrass-nucleus-latest.zip
```

PowerShell

インストールする JDK のバージョンによって、ファイル名が異なる場合があります。インストールした JDK のバージョンに *jdk17.0.6_10* を置き換えてください。

```
'C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe' -
verify -certs -verbose greengrass-nucleus-latest.zip
```

b. `jarsigner` が起動すると、検証結果を示す出力が得られます。

   i. Greengrass nucleus の zip ファイルに署名されると、出力に以下のような文が表示されます：

   ```
   jar verified.
   ```

   ii. Greengrass nucleus の zip ファイルに署名されないと、出力に以下のような文が表示されます：

   ```
   jar is unsigned.
   ```

c. Jarsigner `-certs` を `-verify` と `-verbose` オプションと一緒に提供した場合、出力には署名者証明書の詳細情報も含まれます。

3. AWS IoT Greengrass Core ソフトウェアをデバイス上のフォルダに解凍します。を使用するフォルダ*GreengrassInstaller*に置き換えます。

Linux or Unix

```
unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-
nucleus-latest.zip
```

Windows Command Prompt (CMD)

```
mkdir GreengrassInstaller && tar -xf greengrass-nucleus-latest.zip -
C GreengrassInstaller && del greengrass-nucleus-latest.zip
```

PowerShell

```
Expand-Archive -Path greengrass-nucleus-latest.zip -DestinationPath .\
\GreengrassInstaller
rm greengrass-nucleus-latest.zip
```

4. (オプション) 次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアのバージョンを確認します。

```
java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
```

> **Important**
>
> v2.4.0 以前のバージョンの Greengrass nucleus をインストールする場合、AWS IoT Greengrass Core ソフトウェアをインストール後にこのフォルダを削除しないでください。AWS IoT Greengrass Core ソフトウェアは、このフォルダのファイルを使用して実行します。
> ソフトウェアの最新バージョンをダウンロード済みで、v2.4.0 以降をインストールする場合は、AWS IoT Greengrass Core ソフトウェアをインストールした後にこのフォルダを削除することができます。

AWS IoT Greengrass Core ソフトウェアをインストールします。

次のことを指定する引数を含んだインストーラを実行します。

- コアデバイスの動作に必要な AWS リソースを作成します。
- コアデバイスでソフトウェアコンポーネントを実行するために ggc_user システムユーザーを使用するように指定します。Linux デバイスでは、このコマンドも ggc_group システムグループを使用するように指定し、さらにインストーラによってシステムユーザーとグループが、ユーザーに代わって作成されます。

- AWS IoT Greengrass Core ソフトウェアを、ブート時に実行されるシステムサービスとして設定します。Linux デバイスでは、これは Systemd init システムが必要です。

> **Important**
>
> Windows コアデバイスでは、AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります

ローカル開発ツールで開発デバイスをセットアップするには、`--deploy-dev-tools true` 引数を指定します。インストール完了後、ローカル開発ツールのデプロイには最大 1 分かかることがあります。

指定できる引数の詳細については、「インストーラ引数」を参照してください。

> **Note**
>
> メモリが制限されているデバイスで AWS IoT Greengrass を実行する場合、AWS IoT Greengrass Core ソフトウェアが使用するメモリ量を制御できます。メモリ割り当てを制御するには、nucleus コンポーネントの `jvmOptions` 設定パラメータで JVM ヒープのサイズオプションを設定できます。詳細については、「JVM オプションでメモリ割り当てを制御する」を参照してください。

AWS IoT Greengrass Core ソフトウェアをインストールするには

1. AWS IoT Greengrass コアインストーラを実行します。コマンドの引数値を次のように置き換えます。

   a. */greengrass/v2* または *C:\greengrass\v2*: AWS IoT Greengrass Core ソフトウェアのインストールに使用するルートフォルダへのパス。

   b. *GreengrassInstaller*。AWS IoT Greengrass Core ソフトウェアのインストーラを展開したフォルダへのパス。

   c. *#####*。リソースを検索または作成する AWS リージョン。

   d. *MyGreengrassCore*。Greengrass コアデバイスの AWS IoT モノの名前。モノが存在しない場合、インストーラによって作成されます。インストーラは、証明書をダウンロードして AWS IoT モノとして認証します。詳細については、「AWS IoT Greengrass のデバイス認証と認可」を参照してください。

> **Note**
>
> モノの名前にコロン (:) 記号を含むことができません。

e. *MyGreengrassCoreGroup*。Greengrass コアデバイスの AWS IoT モノグループの名前。モノグループが存在しない場合、インストーラはそのグループを作成してモノを追加します。モノグループが存在してアクティブなデプロイがある場合、コアデバイスはデプロイで指定されたソフトウェアをダウンロードして実行します。

> **Note**
>
> モノグループ名にコロン (:) 記号を含めることはできません。

f. *GreengrassV2IoTThingPolicy#*Greengrass コアデバイスが AWS IoT および AWS IoT Greengrass と通信できるようにする AWS IoT ポリシーの名前。AWS IoT ポリシーが存在しない場合、インストーラが許可を与える AWS IoT ポリシーをこの名前で作成します。ユースケースに合わせて、このポリシーのアクセス許可を制限することができます。詳細については、「AWS IoT Greengrass V2 コアデバイス向けの最低限の AWS IoT ポリシー」を参照してください。

g. *GreengrassV2TokenExchangeRole*。Greengrass コアデバイスが一時的に AWS 認証情報を取得できるようにする IAM ロールの名前。ロールが存在しない場合、インストーラがロールを作成し、*GreengrassV2TokenExchangeRole*Access という名前のポリシーを作成してアタッチします。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

h. *GreengrassCoreTokenExchangeRoleAlias*。Greengrass コアデバイスが後で一時的な認証情報を取得できるようにする IAM ロールのエイリアス。ロールエイリアスが存在しない場合、インストーラがロールエイリアスを作成し、指定した IAM ロールを指します。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

Linux or Unix

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
  -jar ./GreengrassInstaller/lib/Greengrass.jar \
  --aws-region region \
  --thing-name MyGreengrassCore \
```

```
  --thing-group-name MyGreengrassCoreGroup \
  --thing-policy-name GreengrassV2IoTThingPolicy \
  --tes-role-name GreengrassV2TokenExchangeRole \
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias \
  --component-default-user ggc_user:ggc_group \
  --provision true \
  --setup-system-service true
```

Windows Command Prompt (CMD)

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" ^
  -jar ./GreengrassInstaller/lib/Greengrass.jar ^
  --aws-region region ^
  --thing-name MyGreengrassCore ^
  --thing-group-name MyGreengrassCoreGroup ^
  --thing-policy-name GreengrassV2IoTThingPolicy ^
  --tes-role-name GreengrassV2TokenExchangeRole ^
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias ^
  --component-default-user ggc_user ^
  --provision true ^
  --setup-system-service true
```

PowerShell

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" `
  -jar ./GreengrassInstaller/lib/Greengrass.jar `
  --aws-region region `
  --thing-name MyGreengrassCore `
  --thing-group-name MyGreengrassCoreGroup `
  --thing-policy-name GreengrassV2IoTThingPolicy `
  --tes-role-name GreengrassV2TokenExchangeRole `
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias `
  --component-default-user ggc_user `
  --provision true `
  --setup-system-service true
```

Important

Windows コアデバイスでは、`--setup-system-service true` を指定して、AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります

インストーラが正常に処理すると、次のメッセージを印刷します。

- `--provision` を指定すると、インストーラがリソースに正しく設定した場合、`Successfully configured Nucleus with provisioned resource details` を印刷します。
- `--deploy-dev-tools` を指定すると、インストーラがデプロイを正しく作成した場合、`Configured Nucleus to deploy aws.greengrass.Cli component` を印刷します。
- `--setup-system-service true` を指定すると、インストーラがソフトウェアをサービスとして設定して実行した場合、`Successfully set up Nucleus as a system service` を印刷します。
- `--setup-system-service true` を指定しないと、インストーラが正常に処理できてソフトウェアを実行した場合、`Launched Nucleus successfully` を印刷します。

2. Greengrass nucleus v2.0.4 以降をインストールした場合、この手順を省略できます。ソフトウェアの最新バージョンをダウンロード済みで、v2.0.4 以降をインストールしています。

   次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアのルートフォルダに必要なファイル許可を設定します。をインストールコマンドで指定したルートフォルダ*/greengrass/v2*に置き換え、*/greengrass* をルートフォルダの親フォルダに置き換えます。

   ```
   sudo chmod 755 /greengrass/v2 && sudo chmod 755 /greengrass
   ```

AWS IoT Greengrass Core ソフトウェアをシステムサービスとしてインストールした場合、インストーラがソフトウェアを実行します。それ以外の場合、ソフトウェアを手動で実行する必要があります。詳細については、「AWS IoT Greengrass Core ソフトウェアを実行する」を参照してください。

> **Note**
>
> デフォルトでは、インストーラが作成する IAM ロールは S3 バケットのコンポーネントアーティファクトへのアクセスを許可しません。Amazon S3 でアーティファクトを定義するカスタムコンポーネントをデプロイするには、コアデバイスがコンポーネントアーティファクトを取得できるようにする許可をロールに追加する必要があります。詳細については、「コンポーネントのアーティファクトの S3 バケットへのアクセスを許可する」を参照してください。

コンポーネントアーティファクトに S3 バケットをまだ持っていない場合、バケットを作成した後でこれらのアクセス許可を追加できます。

ソフトウェアの設定方法と使用方法、並びに AWS IoT Greengrass の詳細については、次を参照してください。

- AWS IoT Greengrass Core ソフトウェアを設定する
- AWS IoT Greengrass コンポーネントを開発する
- デバイスに AWS IoT Greengrass コンポーネントのデプロイ
- Greengrass コマンドラインインターフェイス

手動リソースプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする

AWS IoT Greengrass Core ソフトウェアには、デバイスを Greengrass コアデバイスとして設定するインストーラが含まれています。デバイスを手動でセットアップするには、デバイスが使用する必須 AWS IoT および IAM リソースを作成できます。これらのリソースを手動で作成する場合、インストーラに AWS 認証情報を提供する必要はありません。

AWS IoT Greengrass Core ソフトウェアを手動でインストールする場合、ネットワークプロキシを使用するか、ポート 443 AWS で に接続するようにデバイスを設定することもできます。例えば、デバイスがファイアウォールやネットワークプロキシの背後で稼働している場合、これらの設定オプションの指定が必要な場合があります。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

PKCS#11 インターフェイス を介してハードウェアセキュリティモジュール (HSM) を使用するように AWS IoT Greengrass Core ソフトウェアを設定することもできます。この機能は、プライベートキーと証明書ファイルを安全に保存して、ソフトウェアで公開または複製されないようにします。プライベートキーと証明書を、HSM、Trusted Platform Module (TPM)、別の暗号化要素などのハードウェアモジュールに保存できます。この機能は、Linux デバイスでのみ利用できます。ハードウェアセキュリティとその使用要件の詳細については、「ハードウェアセキュリティ統合」を参照してください。

> **Important**
>
> AWS IoT Greengrass Core ソフトウェアをダウンロードする前に、コアデバイスが AWS IoT Greengrass Core ソフトウェア v2.0 をインストールして実行するための要件を満たしていることを確認してください。

トピック

AWS IoT エンドポイントを取得する

の AWS IoT エンドポイントを取得し AWS アカウント、後で使用するために保存します。デバイスはこれらのエンドポイントを使用して AWS IoT に接続します。以下の操作を実行します。

1. AWS IoT のデータエンドポイントを取得します AWS アカウント。

   ```
   aws iot describe-endpoint --endpoint-type iot:Data-ATS
   ```

   要求が正常に処理された場合、レスポンスは次の例のようになります。

   ```
   {
     "endpointAddress": "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
   }
   ```

2. の AWS IoT 認証情報エンドポイントを取得します AWS アカウント。

   ```
   aws iot describe-endpoint --endpoint-type iot:CredentialProvider
   ```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "endpointAddress": "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
}
```

AWS IoT モノを作成する

AWS IoT モノは、 に接続するデバイスと論理エンティティを表します AWS IoT。Greengrass コアデバイスは AWS IoT モノです。デバイスを AWS IoT モノとして登録すると、そのデバイスはデジタル証明書を使用して で認証できます AWS。

このセクションでは、デバイスを表す AWS IoT モノを作成します。

AWS IoT モノを作成するには

1. デバイス用の AWS IoT モノを作成します。開発用コンピュータに次のコマンドを実行します。

   - を使用するモノの名前*MyGreengrassCore*に置き換えます。この名前は Greengrass コアデバイスの名前でもあります。

     > **Note**
     >
     > モノの名前にコロン (:) 記号を含むことができません。

   ```
   aws iot create-thing --thing-name MyGreengrassCore
   ```

   要求が正常に処理された場合、レスポンスは次の例のようになります。

   ```
   {
     "thingName": "MyGreengrassCore",
     "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
     "thingId": "8cb4b6cd-268e-495d-b5b9-1713d71dbf42"
   }
   ```

2. （オプション) 新規または既存の AWS IoT モノのグループにモノを追加します。モノグループを使用して Greengrass コアデバイスのフリートを管理します。ソフトウェアコンポーネントをデバイスにデプロイするとき、個々のデバイスまたはデバイスのグループを対象にできます。アクティブな Greengrass デプロイを持つモノグループにデバイスを追加して、そのモノグループのソフトウェアコンポーネントをデバイスにデプロイできます。以下の操作を実行します。

   a. （オプション) AWS IoT モノのグループを作成します。

      - を、作成するモノのグループの名前*MyGreengrassCoreGroup*に置き換えます。

      > **Note**
      >
      > モノグループ名にコロン (:) 記号を含めることはできません。

      ```
      aws iot create-thing-group --thing-group-name MyGreengrassCoreGroup
      ```

      要求が正常に処理された場合、レスポンスは次の例のようになります。

      ```
      {
        "thingGroupName": "MyGreengrassCoreGroup",
        "thingGroupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/
      MyGreengrassCoreGroup",
        "thingGroupId": "4df721e1-ff9f-4f97-92dd-02db4e3f03aa"
      }
      ```

   b. AWS IoT モノをモノグループに追加します。

      - を AWS IoT モノの名前*MyGreengrassCore*に置き換えます。
      - をモノのグループの名前*MyGreengrassCoreGroup*に置き換えます。

      ```
      aws iot add-thing-to-thing-group --thing-name MyGreengrassCore --thing-group-
      name MyGreengrassCoreGroup
      ```

      要求が正常に処理された場合、コマンドは出力されません。

モノの証明書を作成する

デバイスを AWS IoT モノとして登録すると、そのデバイスはデジタル証明書を使用して で認証できます AWS。この証明書により、デバイスは AWS IoT および と通信できます AWS IoT Greengrass。

このセクションでは、デバイスが AWS に接続する際に使用できる証明書を作成してダウンロードします。

ハードウェアセキュリティモジュール (HSM) を使用してプライベートキーと証明書を安全に保存するように AWS IoT Greengrass Core ソフトウェアを設定する場合は、手順に従って HSM のプライベートキーから証明書を作成します。それ以外の場合は、 AWS IoT 手順に従ってサービスに証明書とプライベートキーを作成します。ハードウェアセキュリティ機能は Linux デバイスでのみ利用できます。ハードウェアセキュリティとその使用要件の詳細については、「ハードウェアセキュリティ統合」を参照してください。

AWS IoT サービスで証明書とプライベートキーを作成する

モノの証明書を作成するには

1. AWS IoT モノの証明書をダウンロードするフォルダを作成します。

   ```
   mkdir greengrass-v2-certs
   ```

2. AWS IoT モノの証明書を作成してダウンロードします。

   ```
   aws iot create-keys-and-certificate --set-as-active --certificate-pem-outfile
    greengrass-v2-certs/device.pem.crt --public-key-outfile greengrass-v2-certs/
   public.pem.key --private-key-outfile greengrass-v2-certs/private.pem.key
   ```

   要求が正常に処理された場合、レスポンスは次の例のようになります。

   ```
   {
     "certificateArn": "arn:aws:iot:us-west-2:123456789012:cert/
   aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
     "certificateId":
    "aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
     "certificatePem": "-----BEGIN CERTIFICATE-----
   MIICiTCCAfICCQD6m7oRw0uXOjANBgkqhkiG9w
    0BAQUFADCBiDELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZ
    WF0dGxlMQ8wDQYDVQQKEwZBbWF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIw
   ```

```
 EAYDVQQDEwlUZXN0Q2lsYWMxHzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5
 jb20wHhcNMTEwNDI1MjA0NTIxWhcNMTIwNDI0MjA0NTIxWjCBiDELMAkGA1UEBh
 MCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBb
 WF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIwEAYDVQQDEwlUZXN0Q2lsYWMx
 HzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5jb20wgZ8wDQYJKoZIhvcNAQE
 BBQADgY0AMIGJAoGBAMaK0dn+a4GmWIWJ21uUSfwfEvySWtC2XADZ4nB+BLYgVI
 k60CpiwsZ3G93vUEIO3IyNoH/f0wYK8m9TrDHudUZg3qX4waLG5M43q7Wgc/MbQ
 ITxOUSQv7c7ugFFDzQGBzZswY6786m86gpEIbb3OhjZnzcvQAaRHhdlQWIMm2nr
 AgMBAAEwDQYJKoZIhvcNAQEFBQADgYEAtCu4nUhVVxYUntneD9+h8Mg9q6q+auN
 KyExzyLwaxlAoo7TJHidbtS4J5iNmZgXL0FkbFFBjvSfpJIlJ00zbhNYS5f6Guo
 EDmFJl0ZxBHjJnyp378OD8uTs7fLvjx79LjSTbNYiytVbZPQUQ5Yaxu2jXnimvw
 3rrszlaEXAMPLE=
-----END CERTIFICATE-----",
  "keyPair": {
    "PublicKey": "-----BEGIN PUBLIC KEY-----\
MIIBIjANBgkqhkEXAMPLEQEFAAOCAQ8AMIIBCgKCAQEAEXAMPLE1nnyJwKSMHw4h\
MMEXAMPLEuuN/dMAS3fyce8DW/4+EXAMPLEyjmoF/YVF/gHr99VEEXAMPLE5VF13\
59VK7cEXAMPLE67GK+y+jikqXOgHh/xJTwo
+sGpWEXAMPLEDz18xOd2ka4tCzuWEXAMPLEahJbYkCPUBSU8opVkR7qkEXAMPLE1DR6sx2HocliOOLtu6Fkw91swQWE
\\GB3ZPrNh0PzQYvjUStZeccyNCx2EXAMPLEvp9mQOUXP6plfgxwKRX2fEXAMPLEDa\
hJLXkX3rHU2xbxJSq7D+XEXAMPLEcw+LyFhI5mgFRl88eGdsAEXAMPLElnI9EesG\
FQIDAQAB\
-----END PUBLIC KEY-----\
",
    "PrivateKey": "-----BEGIN RSA PRIVATE KEY-----\
key omitted for security reasons\
-----END RSA PRIVATE KEY-----\
"
  }
}
```

後で証明書を設定するために使用する証明書の Amazon リソースネーム (ARN) を保存します。

HSM のプライベートキーから証明書の作成

> **Note**
>
> この機能は、Greengrass nucleus コンポーネントの v2.5.3 以降で使用できます。 AWS IoT Greengrass 現在、Windows コアデバイスでこの機能をサポートしていません。

モノの証明書を作成するには

1. コアデバイスで、HSM 内に PKCS#11 トークンを初期化してプライベートキーを生成します。プライベートキーは、RSA-2048 キーサイズ (またはそれ以上) の RSA キーまたは ECC キーである必要があります。

   > **Note**
   >
   > ECC キーを備えたハードウェアセキュリティモジュールを使用するには、v2.5.6 以降の Greengrass nucleus を使用する必要があります。
   > ハードウェアセキュリティモジュールとシークレットマネージャーを使用するには、RSA キーを備えたハードウェアセキュリティモジュールを使用する必要があります。

   トークンを初期化してプライベートキーを生成する方法については、HSM のマニュアルを参照してください。HSM がオブジェクト ID をサポートしている場合、プライベートキーの生成時にオブジェクト ID を指定してください。トークンの初期化とプライベートキーの生成時に指定するスロット ID、ユーザー PIN、オブジェクトラベル、オブジェクト ID (HSM が使用する場合) を保存します。これらの値は、後でモノの証明書を HSM にインポートし、 AWS IoT Greengrass Core ソフトウェアを設定するときに使用します。
2. プライベートキーから証明書署名リクエスト (CSR) を作成します。 はこの CSR AWS IoT を使用して、HSM で生成したプライベートキーのモノ証明書を作成します。プライベートキーで CSR を作成する方法については、HSM のマニュアルを参照してください。CSR は `iotdevicekey.csr` などのようなファイルです。
3. CSR をデバイスから開発用コンピュータにコピーします。開発用コンピュータとデバイスで SSH と SCP が有効になっている場合、開発用コンピュータで `scp` コマンドを使用して CSR を転送できます。をデバイスの IP アドレス*device-ip-address*に置き換え、*~/iotdevicekey.csr* をデバイス上の CSR ファイルへのパスに置き換えます。

   ```
   scp device-ip-address:~/iotdevicekey.csr iotdevicekey.csr
   ```
4. 開発用コンピュータで、 AWS IoT モノの証明書をダウンロードするフォルダを作成します。

   ```
   mkdir greengrass-v2-certs
   ```
5. CSR ファイルを使用してモノの証明書を作成し、開発コンピュータ AWS IoT にダウンロードします。

```
aws iot create-certificate-from-csr --set-as-active --certificate-signing-
request=file://iotdevicekey.csr --certificate-pem-outfile greengrass-v2-certs/
device.pem.crt
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "certificateArn": "arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
  "certificateId":
 "aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
  "certificatePem": "-----BEGIN CERTIFICATE-----
MIICiTCCAfICCQD6m7oRw0uXOjANBgkqhkiG9w
 0BAQUFADCBiDELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZ
 WF0dGxlMQ8wDQYDVQQKEwZBbWF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIw
 EAYDVQQDEwlUZXN0Q2lsYWMxHzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5
 jb20wHhcNMTEwNDI1MjA0NTIxWhcNMTIwNDI0MjA0NTIxWjCBiDELMAkGA1UEBh
 MCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBb
 WF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIwEAYDVQQDEwlUZXN0Q2lsYWMx
 HzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5jb20wgZ8wDQYJKoZIhvcNAQE
 BBQADgY0AMIGJAoGBAMaK0dn+a4GmWIWJ21uUSfwfEvySWtC2XADZ4nB+BLYgVI
 k60CpiwsZ3G93vUEIO3IyNoH/f0wYK8m9TrDHudUZg3qX4waLG5M43q7Wgc/MbQ
 ITxOUSQv7c7ugFFDzQGBzZswY6786m86gpEIbb3OhjZnzcvQAaRHhdlQWIMm2nr
 AgMBAAEwDQYJKoZIhvcNAQEFBQADgYEAtCu4nUhVVxYUntneD9+h8Mg9q6q+auN
 KyExzyLwaxlAoo7TJHidbtS4J5iNmZgXL0FkbFFBjvSfpJIlJ00zbhNYS5f6Guo
 EDmFJl0ZxBHjJnyp378OD8uTs7fLvjx79LjSTbNYiytVbZPQUQ5Yaxu2jXnimvw
 3rrszlaEXAMPLE=
-----END CERTIFICATE-----"
}
```

証明書の ARN を保存して後で証明書を設定するために使用します。

モノの証明書を設定する

先ほど AWS IoT 作成したモノにモノの証明書をアタッチし、 ポリシーを証明書に追加 AWS IoT して、コアデバイスの AWS IoT アクセス許可を定義します。

モノの証明書を設定するには

1. 証明書を AWS IoT モノにアタッチします。

- をモノの名前*MyGreengrassCore*に置き換えます AWS IoT 。
- 証明書 Amazon リソースネーム (ARN) を、前のステップで作成した証明書の ARN に置き換えます。

```
aws iot attach-thing-principal --thing-name MyGreengrassCore
 --principal arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
```

要求が正常に処理された場合、コマンドは出力されません。

2. Greengrass コアデバイスの AWS IoT アクセス許可を定義する AWS IoT ポリシーを作成してアタッチします。次のポリシーは、すべての MQTT トピックと Greengrass 操作へのアクセスを許可するため、デバイスがカスタムアプリケーションや新しい Greengrass 操作を必要とする今後の変更でも動作するようになります。ユースケースに基づいてこのポリシーを制限できます。詳細については、「AWS IoT Greengrass V2 コアデバイス向けの最低限の AWS IoT ポリシー」を参照してください。

   Greengrass コアデバイスを以前に設定したことがある場合は、新しいコアデバイスを作成する代わりに、その AWS IoT ポリシーをアタッチできます。

   以下の操作を実行します。

   a. Greengrass コアデバイスが必要とする AWS IoT ポリシードキュメントを含むファイルを作成します。

      例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

      ```
      nano greengrass-v2-iot-policy.json
      ```

      次の JSON をファイルにコピーします。

      ```
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Action": [
      ```

```
        "iot:Publish",
        "iot:Subscribe",
        "iot:Receive",
        "iot:Connect",
        "greengrass:*"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

b. AWS IoT ポリシードキュメントから ポリシーを作成します。

- *GreengrassV2IoTThingPolicy* を作成するポリシーの名前に置き換えます。

```
aws iot create-policy --policy-name GreengrassV2IoTThingPolicy --policy-
document file://greengrass-v2-iot-policy.json
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "policyName": "GreengrassV2IoTThingPolicy",
  "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy",
  "policyDocument": "{
    \\"Version\\": \\"2012-10-17\\",
    \\"Statement\\": [
      {
        \\"Effect\\": \\"Allow\\",
        \\"Action\\": [
          \\"iot:Publish\\",
          \\"iot:Subscribe\\",
          \\"iot:Receive\\",
          \\"iot:Connect\\",
          \\"greengrass:*\\"
        ],
        \\"Resource\\": [
          \\"*\\"
        ]
      }
```

```
    ]
  }",
  "policyVersionId": "1"
}
```

c. AWS IoT モノの証明書に AWS IoT ポリシーをアタッチします。

- *GreengrassV2IoTThingPolicy* をアタッチするポリシーの名前に置き換えます。
- ターゲット ARN を AWS IoT モノの証明書の ARN に置き換えます。

```
aws iot attach-policy --policy-name GreengrassV2IoTThingPolicy
 --target arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
```

要求が正常に処理された場合、コマンドは出力されません。

トークン交換ロールを作成する

Greengrass コアデバイスは、トークン交換ロール と呼ばれる IAM サービスロールを使用して、AWS サービスへの呼び出しを承認します。デバイスは AWS IoT 認証情報プロバイダーを使用して、このロールの一時的な AWS 認証情報を取得します。これにより、デバイスは とやり取りし AWS IoT、Amazon CloudWatch Logs にログを送信し、Amazon S3 からカスタムコンポーネントアーティファクトをダウンロードできるようになります。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

AWS IoT ロールエイリアスを使用して、Greengrass コアデバイスのトークン交換ロールを設定します。ロールエイリアスは、デバイスのトークン交換ロールを変更できるようにしますが、デバイス設定は同じ内容に保たれます。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS サービスへの直接呼び出しを認証する」を参照してください。

このセクションでは、トークン交換 IAM ロールと、 AWS IoT そのロールを指す ロールエイリアスを作成します。Greengrass コアデバイスを既に設定している場合、新しく作成せず、トークン交換ロールとロールエイリアスを使用できます。次に、デバイスの AWS IoT モノを設定してそのロールとエイリアスを使用します。

トークン交換 IAM ロールを作成するには

1. デバイスがトークン交換ロールとして使用できる IAM ロールを作成します。以下の操作を実行します。

a. トークン交換ロールが必要とする、信頼できるポリシードキュメントが含まれるファイルを作成します。

例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano device-role-trust-policy.json
```

次の JSON をファイルにコピーします。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.iot.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

b. 信頼ポリシードキュメントでトークン交換ロールを作成します。

- *GreengrassV2TokenExchangeRole* を作成する IAM ロールの名前に置き換えます。

```
aws iam create-role --role-name GreengrassV2TokenExchangeRole --assume-role-
policy-document file://device-role-trust-policy.json
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "Role": {
    "Path": "/",
    "RoleName": "GreengrassV2TokenExchangeRole",
    "RoleId": "AROAZ2YMUHYHK5OKM77FB",
    "Arn": "arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole",
    "CreateDate": "2021-02-06T00:13:29+00:00",
    "AssumeRolePolicyDocument": {
```

```
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "credentials.iot.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
  }
```

c. トークン交換ロールが必要なアクセスポリシードキュメントを含むファイルを作成します。

例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano device-role-access-policy.json
```

次の JSON をファイルにコピーします。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
}
```

> **Note**
>
> このアクセスポリシーでは、S3 バケットのコンポーネントアーティファクトへのアクセスが許可されていません。Amazon S3 でアーティファクトを定義するカスタムコンポーネントをデプロイするには、コアデバイスがコンポーネントアーティファクトを取得できるようにする許可をロールに追加する必要があります。詳細については、「コンポーネントのアーティファクトの S3 バケットへのアクセスを許可する」を参照してください。
> コンポーネントアーティファクトに S3 バケットをまだ持っていない場合、バケットを作成した後でこれらのアクセス許可を追加できます。

d. ポリシードキュメントから IAM ポリシーを作成します。

- *GreengrassV2TokenExchangeRoleAccess* を作成する IAM ポリシーの名前に置き換えます。

```
aws iam create-policy --policy-name GreengrassV2TokenExchangeRoleAccess --
policy-document file://device-role-access-policy.json
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "Policy": {
    "PolicyName": "GreengrassV2TokenExchangeRoleAccess",
    "PolicyId": "ANPAZ2YMUHYHACI7C5Z66",
    "Arn": "arn:aws:iam::123456789012:policy/
GreengrassV2TokenExchangeRoleAccess",
    "Path": "/",
    "DefaultVersionId": "v1",
    "AttachmentCount": 0,
    "PermissionsBoundaryUsageCount": 0,
    "IsAttachable": true,
    "CreateDate": "2021-02-06T00:37:17+00:00",
    "UpdateDate": "2021-02-06T00:37:17+00:00"
  }
}
```

e. IAM ポリシーをトークン交換ロールにアタッチします。

- *GreengrassV2TokenExchangeRole* を IAM ロールの名前に置き換えます。
- ポリシー ARN を前のステップで作成した IAM ポリシーの ARN に置き換えます。

```
aws iam attach-role-policy --role-name GreengrassV2TokenExchangeRole --policy-
arn arn:aws:iam::123456789012:policy/GreengrassV2TokenExchangeRoleAccess
```

要求が正常に処理された場合、コマンドは出力されません。

2. トークン交換 AWS IoT ロールを指す ロールエイリアスを作成します。

- を、作成するロールエイリアスの名前*GreengrassCoreTokenExchangeRoleAlias*に置き換えます。
- ロール ARN を前のステップで作成した IAM ロールの ARN に置き換えます。

```
aws iot create-role-alias --role-alias GreengrassCoreTokenExchangeRoleAlias --role-
arn arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "roleAlias": "GreengrassCoreTokenExchangeRoleAlias",
  "roleAliasArn": "arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias"
}
```

> **Note**
>
> ロールエイリアスを作成するには、トークン交換 IAM ロールを AWS IoT に渡す許可が必要です。ロールエイリアスを作成しようとしたときにエラーメッセージが表示された場合は、 AWS ユーザーにこのアクセス許可があることを確認します。詳細については、「 ユーザーガイド」の「 AWS サービスにロールを渡すアクセス許可をユーザーに付与する」を参照してください。 AWS Identity and Access Management

3. Greengrass コアデバイスがロールエイリアスを使用してトークン交換ロールを引き受けることを許可する AWS IoT ポリシーを作成してアタッチします。Greengrass コアデバイスを以前に

設定したことがある場合は、新しいロールエイリアスポリシーを作成する代わりに、そのロールエイリアス AWS IoT ポリシーをアタッチできます。以下の操作を実行します。

a. (オプション) ロールエイリアスに必要な AWS IoT ポリシードキュメントを含むファイルを作成します。

例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano greengrass-v2-iot-role-alias-policy.json
```

次の JSON をファイルにコピーします。

- リソース ARN をロールエイリアスの ARN に置き換えます。

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:AssumeRoleWithCertificate",
      "Resource": "arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias"
    }
  ]
}
```

b. AWS IoT ポリシードキュメントから ポリシーを作成します。

- を作成する AWS IoT ポリシーの名前*GreengrassCoreTokenExchangeRoleAliasPolicy*に置き換えます。

```
aws iot create-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy
 --policy-document file://greengrass-v2-iot-role-alias-policy.json
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "policyName": "GreengrassCoreTokenExchangeRoleAliasPolicy",
```

```
  "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassCoreTokenExchangeRoleAliasPolicy",
  "policyDocument": "{
    \\"Version\\":\\"2012-10-17\\",
    \\"Statement\\": [
      {
        \\"Effect\\": \\"Allow\\",
        \\"Action\\": \\"iot:AssumeRoleWithCertificate\\",
        \\"Resource\\": \\"arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias\\"
      }
    ]
  }",
  "policyVersionId": "1"
}
```

c. AWS IoT モノの証明書に AWS IoT ポリシーをアタッチします。

- をロールエイリアス AWS IoT ポリシーの名前*GreengrassCoreTokenExchangeRoleAliasPolicy*に置き換えます。
- ターゲット ARN を AWS IoT モノの証明書の ARN に置き換えます。

```
aws iot attach-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy
 --target arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
```

要求が正常に処理された場合、コマンドは出力されません。

デバイスに証明書をダウンロードする

以前に、デバイスの証明書を開発用コンピュータにダウンロードしました。このセクションでは、コアデバイスに証明書をコピーして、 AWS IoT に接続するために使用する証明書でデバイスをセットアップします。Amazon ルート認証機関 (CA) の証明書もダウンロードします。HSM を使用する場合、このセクションの HSM に証明書ファイルもインポートします。

- 以前に AWS IoT サービスでモノの証明書とプライベートキーを作成した場合は、手順に従ってプライベートキーと証明書ファイルを含む証明書をダウンロードします。
- 以前にハードウェアセキュリティモジュール (HSM) のプライベートキーからモノ証明書を作成した場合、手順に従って HSM のプライベートキーと証明書で証明書をダウンロードします。

プライベートキーと証明書ファイルで証明書をダウンロード

証明書をデバイスにダウンロードするには

1. 開発用コンピュータからデバイスに AWS IoT モノの証明書をコピーします。開発用コンピュータとデバイスで SSH と SCP が有効になっている場合、開発用コンピュータの `scp` コマンドを実行して、証明書を転送できます。をデバイスの IP アドレス*device-ip-address*に置き換えます。

   ```
   scp -r greengrass-v2-certs/ device-ip-address:~
   ```

2. デバイスに Greengrass ルートフォルダを作成します。後で AWS IoT Greengrass Core ソフトウェアをこのフォルダにインストールします。

   Linux or Unix

   - を使用するフォルダ*/greengrass/v2*に置き換えます。

   ```
   sudo mkdir -p /greengrass/v2
   ```

   Windows Command Prompt

   - *C:\greengrass\v2* を使用するフォルダに置き換えます。

   ```
   mkdir C:\greengrass\v2
   ```

   PowerShell

   - *C:\greengrass\v2* を使用するフォルダに置き換えます。

   ```
   mkdir C:\greengrass\v2
   ```

3. (Linux のみ) Greengrass ルートフォルダの親の許可を設定します。

   - */greengrass* をルートフォルダへの親に置き換えます。

   ```
   sudo chmod 755 /greengrass
   ```

4. AWS IoT モノの証明書を Greengrass ルートフォルダにコピーします。

   Linux or Unix

   - を Greengrass ルートフォルダ*/greengrass/v2*に置き換えます。

   ```
   sudo cp -R ~/greengrass-v2-certs/* /greengrass/v2
   ```

   Windows Command Prompt

   - *C:\greengrass\v2* を使用するフォルダに置き換えます。

   ```
   robocopy %USERPROFILE%\greengrass-v2-certs C:\greengrass\v2 /E
   ```

   PowerShell

   - *C:\greengrass\v2* を使用するフォルダに置き換えます。

   ```
   cp -Path ~\greengrass-v2-certs\* -Destination C:\greengrass\v2
   ```

5. Amazon ルート認証局 (CA) certificate. AWS IoT certificates は、デフォルトで Amazon のルート CA 証明書に関連付けられています。

   Linux or Unix

   ```
   sudo curl -o /greengrass/v2/AmazonRootCA1.pem https://www.amazontrust.com/
   repository/AmazonRootCA1.pem
   ```

   Windows Command Prompt (CMD)

   ```
   curl -o C:\greengrass\v2\\AmazonRootCA1.pem https://www.amazontrust.com/
   repository/AmazonRootCA1.pem
   ```

   PowerShell

   ```
   iwr -Uri https://www.amazontrust.com/repository/AmazonRootCA1.pem -OutFile C:
   \greengrass\v2\\AmazonRootCA1.pem
   ```

HSM にプライベートキーと証明書を含む証明書をダウンロード

> **Note**
>
> この機能は、Greengrass nucleus コンポーネントの v2.5.3 以降で使用できます。 AWS IoT Greengrass 現在、Windows コアデバイスでこの機能をサポートしていません。

証明書をデバイスにダウンロードするには

1. 開発用コンピュータからデバイスに AWS IoT モノの証明書をコピーします。開発用コンピュータとデバイスで SSH と SCP が有効になっている場合、開発用コンピュータの `scp` コマンドを実行して、証明書を転送できます。をデバイスの IP アドレス*device-ip-address*に置き換えます。

   ```
   scp -r greengrass-v2-certs/ device-ip-address:~
   ```

2. デバイスに Greengrass ルートフォルダを作成します。後で AWS IoT Greengrass Core ソフトウェアをこのフォルダにインストールします。

   Linux or Unix

   - を使用するフォルダ*/greengrass/v2*に置き換えます。

   ```
   sudo mkdir -p /greengrass/v2
   ```

   Windows Command Prompt

   - *C:\greengrass\v2* を使用するフォルダに置き換えます。

   ```
   mkdir C:\greengrass\v2
   ```

   PowerShell

   - *C:\greengrass\v2* を使用するフォルダに置き換えます。

   ```
   mkdir C:\greengrass\v2
   ```

3. (Linux のみ) Greengrass ルートフォルダの親の許可を設定します。

   - */greengrass* をルートフォルダへの親に置き換えます。

   ```
   sudo chmod 755 /greengrass
   ```

4. モノの証明書ファイルである ~/greengrass-v2-certs/device.pem.crt を HSM にインポートします。証明書をインポートする方法については、HSM のマニュアルを参照してください。以前に HSM でプライベートキーを生成したときと同じトークン、スロット ID、ユーザー PIN、オブジェクトラベル、オブジェクト ID (HSM が使用している場合) を使用して証明書をインポートします。

   > **Note**
   >
   > 以前にオブジェクト ID なしでプライベートキーを生成し、かつ証明書にオブジェクト ID がある場合、プライベートキーのオブジェクト ID を証明書と同じ値に設定します。プライベートキーオブジェクトのオブジェクト ID を設定する方法については、HSM のマニュアルを参照してください。

5. (オプション) モノ証明書のファイルを削除して、HSM にのみ存在するようにします。

   ```
   rm ~/greengrass-v2-certs/device.pem.crt
   ```

6. Amazon ルート認証局 (CA) certificate. AWS IoT certificates は、デフォルトで Amazon のルート CA 証明書に関連付けられています。

   Linux or Unix

   ```
   sudo curl -o /greengrass/v2/AmazonRootCA1.pem https://www.amazontrust.com/
   repository/AmazonRootCA1.pem
   ```

   Windows Command Prompt (CMD)

   ```
   curl -o C:\greengrass\v2\\AmazonRootCA1.pem https://www.amazontrust.com/
   repository/AmazonRootCA1.pem
   ```

PowerShell

```
iwr -Uri https://www.amazontrust.com/repository/AmazonRootCA1.pem -OutFile C:
\greengrass\v2\\AmazonRootCA1.pem
```

デバイス環境をセットアップする

このセクションのステップに従って、 AWS IoT Greengrass コアデバイスとして使用する Linux または Windows デバイスをセットアップします。

Linux デバイスをセットアップする

の Linux デバイスをセットアップするには AWS IoT Greengrass V2

1. AWS IoT Greengrass Core ソフトウェアが実行する必要がある Java ランタイムをインストールします。Amazon Corretto または OpenJDK の長期サポートバージョンを使用することをお勧めします。バージョン 8 以降が必要です。次のコマンドは、デバイスに OpenJDK をインストールする方法を示しています。

   - Debian ベースまたは Ubuntu ベースのディストリビューションの場合:

     ```
     sudo apt install default-jdk
     ```

   - Red Hat ベースのディストリビューションの場合 :

     ```
     sudo yum install java-11-openjdk-devel
     ```

   - 複数 Amazon Linux 2:

     ```
     sudo amazon-linux-extras install java-openjdk11
     ```

   - 複数 Amazon Linux 2023:

     ```
     sudo dnf install java-11-amazon-corretto -y
     ```

   インストールが完了したら、次のコマンドを実行して Java が Linux デバイスで実行されていることを確認します。

```
java -version
```

このコマンドは、デバイス上で実行されている Java のバージョンを出力します。例えば、Debian ベースのディストリビューションでは、出力は次のサンプルのようになります。

```
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

2. (オプション) デバイスにコンポーネントを実行するデフォルトのシステムユーザーおよびグループを作成します。インストーラ引数を使用して、インストール中に AWS IoT Greengrass Core ソフトウェアインストーラでこのユーザーとグループを作成させることもできます。詳細については、「インストーラ引数」を参照してください。

```
sudo useradd --system --create-home ggc_user
sudo groupadd --system ggc_group
```

3. AWS IoT Greengrass Core ソフトウェアを実行するユーザー (通常は `root`) に、任意のユーザーおよび任意のグループで実行するアクセス許可があることを確認します。

   a. `/etc/sudoers` ファイルを開くには、次のコマンドを実行します。

   ```
   sudo visudo
   ```

   b. ユーザーの権限が次の例のようになっていることを確認します。

   ```
   root    ALL=(ALL:ALL) ALL
   ```

4. (オプション) コンテナ化された Lambda 関数を実行するには、cgroups v1 を有効にし、メモリとデバイスの cgroups を有効にしてマウントする必要があります。コンテナ化された Lambda 関数を実行する予定がない場合、この手順を省略できます。

   これらの cgroups オプションを有効にするには、次の Linux カーネルパラメータを使用してデバイスを起動します。

   ```
   cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
   ```

デバイスのカーネルパラメータを確認および設定するための情報については、オペレーティングシステムおよびブートローダーのドキュメントを参照してください。指示に従って、カーネルパラメータを永続的に設定します。

5. デバイスの要件 にある要件リストで示されているように、その他の必要となる依存関係をすべてデバイスにインストールします。

Windows デバイスをセットアップする

> **Note**
>
> この機能は、Greengrass nucleus コンポーネントの v2.5.0 以降に利用できます。

用の Windows デバイスを設定するには AWS IoT Greengrass V2

1. AWS IoT Greengrass Core ソフトウェアが実行する必要がある Java ランタイムをインストールします。Amazon Corretto または OpenJDK の長期サポートバージョンを使用することをお勧めします。バージョン 8 以降が必要です。
2. PATH システム変数で Java が使用可能か確認し、そうでない場合は追加します。 LocalSystem アカウントは AWS IoT Greengrass Core ソフトウェアを実行するため、ユーザーの PATH ユーザー変数の代わりに Java を PATH システム変数に追加する必要があります。以下の操作を実行します。

   a. Windows キーを押してスタートメニューを開きます。

   b. **`environment variables`** を入力して、スタートメニューからシステムオプションを検索します。

   c. スタートメニューの検索結果から [Edit the system environment variables] (システム環境変数を編集) をクリックして、[System properties] (システムプロパティ) ウィンドウを開きます。

   d. [Environment variables...] (環境変数...) を選択して、[Environment Variables] (環境可変) ウィンドウを開きます。

   e. [System variables] (システム変数) で、[Path] (パス) 、[Edit] (編集) の順に選択します。[Edit environment variable] (環境変数の編集) ウィンドウでは、個別の行に各パスを表示できます。

f. Java インストールの bin フォルダへのパスが存在しているかを確認します。このパスは、次の例のように表示されます。

```
C:\\Program Files\\Amazon Corretto\\jdk11.0.13_8\\bin
```

g. [Path] (パス) で Java インストールの bin フォルダが で見つからない場合は、[New] (新規) を選択してこれを追加した上で、[OK] を選択します。

3. 管理者として Windows コマンドプロンプト cmd.exe を開きます。

4. Windows デバイスの LocalSystem アカウントにデフォルトユーザーを作成します。*#####*を安全なパスワードに置き換えます。

```
net user /add ggc_user password
```

> **Tip**
>
> Windows の構成によっては、ユーザーのパスワードの期限切れが、将来の日付に設定されている場合があります。Greengrass アプリケーションの動作を継続させるためには、パスワードの有効期限を追跡し、その期限が切れる前に更新します。ユーザーのパスワードには、期限切れを起こさないような設定も可能です。
>
> - ユーザーとパスワードの有効期限を確認するには、次のコマンドを実行します。
>
> ```
> net user ggc_user | findstr /C:expires
> ```
>
> - ユーザーのパスワードが期限切れにならないように設定するには、次のコマンドを実行します。
>
> ```
> wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
> ```
>
> - wmic コマンドが廃止された Windows 10 以降を使用している場合は、次の PowerShell コマンドを実行します。
>
> ```
> Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
>  'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
> ```

5. デバイスに Microsoft からPsExecユーティリティをダウンロードしてインストールします。

6. PsExec ユーティリティを使用して、デフォルトユーザーのユーザー名とパスワードを LocalSystem アカウントの認証情報マネージャーインスタンスに保存します。*#####*を以前に設定したユーザーのパスワードに置き換えます。

   ```
   psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
   ```

   PsExec License Agreement が開いたら、Accept を選択し、ライセンスに同意してコマンドを実行します。

   > **Note**
   >
   > Windows デバイスでは、 LocalSystem アカウントが Greengrass nucleus を実行するため、 PsExec ユーティリティを使用してデフォルトのユーザー情報を LocalSystem アカウントに格納する必要があります。認証情報マネージャーアプリケーションを使用すると、この情報は アカウントではなく、現在ログオンしているユーザーの Windows LocalSystem アカウントに保存されます。

AWS IoT Greengrass Core ソフトウェアをダウンロードする

AWS IoT Greengrass Core ソフトウェアの最新バージョンは、次の場所からダウンロードできます。

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアの特定のバージョンは、次の場所からダウンロードできます。*#####*をダウンロードするバージョンに置き換えます。
>
> ```
> https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip
> ```

AWS IoT Greengrass Core ソフトウェアをダウンロードするには

1. コアデバイスで、 AWS IoT Greengrass Core ソフトウェアを という名前のファイルにダウンロードしますgreengrass-nucleus-latest.zip。

Linux or Unix

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip > greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip > greengrass-nucleus-latest.zip
```

PowerShell

```
iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip -OutFile greengrass-nucleus-latest.zip
```

このソフトウェアをダウンロードすると、Greengrass Core ソフトウェアのライセンス契約に同意したものと見なされます。

2. (オプション) Greengrass nucleus ソフトウェア署名を確認するには

> **Note**
>
> この機能は、Greengrass nucleus バージョン 2.9.5 以降で使用できます。

a. 以下のコマンドを使用して、Greengrass nucleus アーティファクトの署名を確認します。

Linux or Unix

```
jarsigner -verify -certs -verbose greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

インストールする JDK のバージョンによって、ファイル名が異なる場合があります。インストールした JDK のバージョンに *jdk17.0.6_10* を置き換えてください。

```
"C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe" -
verify -certs -verbose greengrass-nucleus-latest.zip
```

PowerShell

インストールする JDK のバージョンによって、ファイル名が異なる場合があります。インストールした JDK のバージョンに *jdk17.0.6_10* を置き換えてください。

```
'C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe' -
verify -certs -verbose greengrass-nucleus-latest.zip
```

b. `jarsigner` が起動すると、検証結果を示す出力が得られます。

i. Greengrass nucleus の zip ファイルに署名されると、出力に以下のような文が表示されます：

```
jar verified.
```

ii. Greengrass nucleus の zip ファイルに署名されないと、出力に以下のような文が表示されます：

```
jar is unsigned.
```

c. Jarsigner `-certs` を `-verify` と `-verbose` オプションと一緒に提供した場合、出力には署名者証明書の詳細情報も含まれます。

3. AWS IoT Greengrass Core ソフトウェアをデバイス上のフォルダに解凍します。を使用するフォルダ*GreengrassInstaller*に置き換えます。

Linux or Unix

```
unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-
nucleus-latest.zip
```

Windows Command Prompt (CMD)

```
mkdir GreengrassInstaller && tar -xf greengrass-nucleus-latest.zip -
C GreengrassInstaller && del greengrass-nucleus-latest.zip
```

PowerShell

```
Expand-Archive -Path greengrass-nucleus-latest.zip -DestinationPath .\
\GreengrassInstaller
```

```
rm greengrass-nucleus-latest.zip
```

4. (オプション) 次のコマンドを実行して、 AWS IoT Greengrass Core ソフトウェアのバージョンを確認します。

```
java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
```

> **Important**
>
> v2.4.0 より前のバージョンの Greengrass nucleus をインストールする場合は、 AWS IoT Greengrass Core ソフトウェアをインストールした後にこのフォルダを削除しないでください。 AWS IoT Greengrass Core ソフトウェアは、このフォルダ内のファイルを使用して実行します。
> ソフトウェアの最新バージョンをダウンロードした場合は、v2.4.0 以降をインストールし、AWS IoT Greengrass Core ソフトウェアをインストールした後にこのフォルダを削除できます。

AWS IoT Greengrass Core ソフトウェアのインストール

次のアクションを指定する引数を含むインストーラを実行します。

- 以前に作成した AWS リソースと証明書を使用するように指定する部分設定ファイルからインストールします。 AWS IoT Greengrass Core ソフトウェアは、デバイス上のすべての Greengrass コンポーネントの設定を指定する設定ファイルを使用します。インストーラは、提供する部分設定ファイルで完全な設定ファイルを作成します。
- コアデバイスでソフトウェアコンポーネントを実行するために `ggc_user` システムユーザーを使用するように指定します。Linux デバイスでは、このコマンドも `ggc_group` システムグループを使用するように指定し、さらにインストーラによってシステムユーザーとグループが、ユーザーに代わって作成されます。
- AWS IoT Greengrass Core ソフトウェアを、起動時に実行されるシステムサービスとしてセットアップします。Linux デバイスでは、これは Systemd init システムが必要です。

> **Important**
>
> Windows コアデバイスでは、 AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります。

指定できる引数の詳細については、「インストーラ引数」を参照してください。

> **Note**
>
> メモリが制限された AWS IoT Greengrass デバイスで実行している場合は、 AWS IoT Greengrass Core ソフトウェアが使用するメモリ量を制御できます。メモリ割り当てを制御するには、nucleus コンポーネントの `jvmOptions` 設定パラメータで JVM ヒープのサイズオプションを設定できます。詳細については、「JVM オプションでメモリ割り当てを制御する」を参照してください。

- 以前に AWS IoT サービスでモノの証明書とプライベートキーを作成した場合は、手順に従ってプライベートキーと証明書ファイルを含む AWS IoT Greengrass Core ソフトウェアをインストールします。
- 以前にハードウェアセキュリティモジュール (HSM) のプライベートキーからモノの証明書を作成した場合は、手順に従って、プライベートキーと証明書を含む AWS IoT Greengrass Core ソフトウェアを HSM にインストールします。

プライベートキーと証明書ファイルを使用して AWS IoT Greengrass Core ソフトウェアをインストールする

AWS IoT Greengrass Core ソフトウェアをインストールするには

1. AWS IoT Greengrass Core ソフトウェアのバージョンを確認します。

   - を、ソフトウェアを含むフォルダへのパス*GreengrassInstaller*に置き換えます。

   ```
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

2. テキストエディタを使用し、`config.yaml` という名前の設定ファイルを作成してインストーラに提供します。

例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano GreengrassInstaller/config.yaml
```

次の YAML コンテンツをファイルにコピーします。この部分設定ファイルは、システムパラメータと Greengrass nucleus パラメータを指定します。

```
---
system:
  certificateFilePath: "/greengrass/v2/device.pem.crt"
  privateKeyPath: "/greengrass/v2/private.pem.key"
  rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
  rootpath: "/greengrass/v2"
  thingName: "MyGreengrassCore"
services:
  aws.greengrass.Nucleus:
    componentType: "NUCLEUS"
    version: "2.12.2"
    configuration:
      awsRegion: "us-west-2"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      iotCredEndpoint: "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
```

次に、以下の操作を実行します。

- の各インスタンスを Greengrass ルートフォルダ*/greengrass/v2*に置き換えます。
- をモノの名前*MyGreengrassCore*に置き換えます AWS IoT 。
- *2.12.2* を AWS IoT Greengrass Core ソフトウェアのバージョンに置き換えます。
- *us-west-2* を、リソースを作成した AWS リージョン に置き換えます。
- をトークン交換ロールエイリアスの名前*GreengrassCoreTokenExchangeRoleAlias*に置き換えます。
- を AWS IoT データエンドポイント`iotDataEndpoint`に置き換えます。
- を AWS IoT 認証情報エンドポイント`iotCredEndpoint`に置き換えます。

> **Note**
>
> この設定ファイルでは、使用するポートやネットワークプロキシなど、次の例で示すように、他の nucleus の設定オプションをカスタマイズできます。詳細については、Greengrass nucleus 設定を参照してください。

```
---
system:
  certificateFilePath: "/greengrass/v2/device.pem.crt"
  privateKeyPath: "/greengrass/v2/private.pem.key"
  rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
  rootpath: "/greengrass/v2"
  thingName: "MyGreengrassCore"
services:
  aws.greengrass.Nucleus:
    componentType: "NUCLEUS"
    version: "2.12.2"
    configuration:
      awsRegion: "us-west-2"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      iotCredEndpoint: "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      mqtt:
        port: 443
      greengrassDataPlanePort: 443
      networkProxy:
        noProxyAddresses: "http://192.168.0.1,www.example.com"
        proxy:
          url: "https://my-proxy-server:1100"
          username: "Mary_Major"
          password: "pass@word1357"
```

3. インストーラを実行して、設定ファイルを提供するように、`--init-config` に対して指定します。

   - */greengrass/v2* または *C:\greengrass\v2* を Greengrass ルートフォルダに置き換えます。

- の各インスタンスを、インストーラを解凍したフォルダ*GreengrassInstaller*に置き換えます。

Linux or Unix

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
  -jar ./GreengrassInstaller/lib/Greengrass.jar \
  --init-config ./GreengrassInstaller/config.yaml \
  --component-default-user ggc_user:ggc_group \
  --setup-system-service true
```

Windows Command Prompt (CMD)

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" ^
  -jar ./GreengrassInstaller/lib/Greengrass.jar ^
  --init-config ./GreengrassInstaller/config.yaml ^
  --component-default-user ggc_user ^
  --setup-system-service true
```

PowerShell

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" `
  -jar ./GreengrassInstaller/lib/Greengrass.jar `
  --init-config ./GreengrassInstaller/config.yaml `
  --component-default-user ggc_user `
  --setup-system-service true
```

Important

Windows コアデバイスでは、 を指定`--setup-system-service true`して AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります。

`--setup-system-service true` を指定する場合、ソフトウェアをシステムサービスとしてセットアップして実行したら、インストーラは `Successfully set up Nucleus as a system service` を出力します。それ以外の場合、正常にソフトウェアがインストールされていれば、インストーラはメッセージを出力しません。

> **Note**
>
> `--provision true` 引数なしでインストーラを実行するとき、ローカル開発ツールをデプロイするために `deploy-dev-tools` 引数を使用できません。Greengrass CLI をデバイスに直接デプロイする方法の情報については、「Greengrass コマンドラインインターフェイス」を参照してください。

4. ルートフォルダのファイルを確認して、インストールを確認します。

   Linux or Unix

   ```
   ls /greengrass/v2
   ```

   Windows Command Prompt (CMD)

   ```
   dir C:\greengrass\v2
   ```

   PowerShell

   ```
   ls C:\greengrass\v2
   ```

   インストールが正常に処理された場合、ルートフォルダには `config`、`packages`、`logs` などの複数のフォルダが含まれます。

プライベートキーと証明書を使用して AWS IoT Greengrass Core ソフトウェアを HSM にインストールする

> **Note**
>
> この機能は、Greengrass nucleus コンポーネントの v2.5.3 以降で使用できます。 AWS IoT Greengrass 現在、Windows コアデバイスでこの機能をサポートしていません。

AWS IoT Greengrass Core ソフトウェアをインストールするには

1. AWS IoT Greengrass Core ソフトウェアのバージョンを確認します。

- を、ソフトウェアを含むフォルダへのパス*GreengrassInstaller*に置き換えます。

```
java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
```

2. AWS IoT Greengrass Core ソフトウェアが HSM でプライベートキーと証明書を使用できるようにするには、 AWS IoT Greengrass Core ソフトウェアをインストールするときに PKCS#11 プロバイダコンポーネントをインストールします。PKCS#11 プロバイダコンポーネントは、インストール時に設定できるプラグインです。PKCS#11 プロバイダコンポーネントの最新バージョンを次の場所からダウンロードできます。

   - https://d2s8p88vqu9w66.cloudfront.net/releases/Pkcs11Provider/aws.greengrass.crypto.Pkcs11Provider-latest.jar

   PKCS#11 プロバイダプラグインを aws.greengrass.crypto.Pkcs11Provider.jar という名前のファイルにダウンロードします。を使用するフォルダ*GreengrassInstaller*に置き換えます。

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/Pkcs11Provider/
   aws.greengrass.crypto.Pkcs11Provider-latest.jar > GreengrassInstaller/
   aws.greengrass.crypto.Pkcs11Provider.jar
   ```

   このソフトウェアをダウンロードすると、Greengrass Core ソフトウェアのライセンス契約に同意したものと見なされます。

3. テキストエディタを使用し、config.yaml という名前の設定ファイルを作成してインストーラに提供します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

   ```
   nano GreengrassInstaller/config.yaml
   ```

   次の YAML コンテンツをファイルにコピーします。この部分設定ファイルは、システムパラメータ、Greengrass nucleus パラメータ、PKCS#11 プロバイダパラメータを指定します。

   ```
   ---
   system:
   ```

```
  certificateFilePath: "pkcs11:object=iotdevicekey;type=cert"
  privateKeyPath: "pkcs11:object=iotdevicekey;type=private"
  rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
  rootpath: "/greengrass/v2"
  thingName: "MyGreengrassCore"
services:
  aws.greengrass.Nucleus:
    componentType: "NUCLEUS"
    version: "2.12.2"
    configuration:
      awsRegion: "us-west-2"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      iotCredEndpoint: "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
  aws.greengrass.crypto.Pkcs11Provider:
    configuration:
      name: "softhsm_pkcs11"
      library: "/usr/local/Cellar/softhsm/2.6.1/lib/softhsm/libsofthsm2.so"
      slot: 1
      userPin: "1234"
```

次に、以下の操作を実行します。

- PKCS#11 URI の *iotdevicekey* の各インスタンスを、プライベートキーを作成して証明書をインポートしたオブジェクトラベルに置き換えます。
- の各インスタンスを Greengrass ルートフォルダ*/greengrass/v2*に置き換えます。
- をモノの名前*MyGreengrassCore*に置き換えます AWS IoT 。
- *2.12.2* を AWS IoT Greengrass Core ソフトウェアのバージョンに置き換えます。
- *us-west-2* を、リソースを作成した AWS リージョン に置き換えます。
- をトークン交換ロールエイリアスの名前*GreengrassCoreTokenExchangeRoleAlias*に置き換えます。
- を AWS IoT データエンドポイント`iotDataEndpoint`に置き換えます。
- を AWS IoT 認証情報エンドポイント`iotCredEndpoint`に置き換えます。
- `aws.greengrass.crypto.Pkcs11Provider` コンポーネントの設定パラメータを、コアデバイスの HSM 設定の値に置き換えます。

Note

この設定ファイルでは、使用するポートやネットワークプロキシなど、次の例で示すように、他の nucleus の設定オプションをカスタマイズできます。詳細については、Greengrass nucleus 設定を参照してください。

```
---
system:
  certificateFilePath: "pkcs11:object=iotdevicekey;type=cert"
  privateKeyPath: "pkcs11:object=iotdevicekey;type=private"
  rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
  rootpath: "/greengrass/v2"
  thingName: "MyGreengrassCore"
services:
  aws.greengrass.Nucleus:
    componentType: "NUCLEUS"
    version: "2.12.2"
    configuration:
      awsRegion: "us-west-2"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      iotCredEndpoint: "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
      mqtt:
        port: 443
      greengrassDataPlanePort: 443
      networkProxy:
        noProxyAddresses: "http://192.168.0.1,www.example.com"
        proxy:
          url: "https://my-proxy-server:1100"
          username: "Mary_Major"
          password: "pass@word1357"
  aws.greengrass.crypto.Pkcs11Provider:
    configuration:
      name: "softhsm_pkcs11"
      library: "/usr/local/Cellar/softhsm/2.6.1/lib/softhsm/libsofthsm2.so"
      slot: 1
      userPin: "1234"
```

4. インストーラを実行して、設定ファイルを提供するように、--init-config に対して指定します。

   - を Greengrass ルートフォルダ*/greengrass/v2*に置き換えます。
   - の各インスタンスを、インストーラを解凍したフォルダ*GreengrassInstaller*に置き換えます。

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
  -jar ./GreengrassInstaller/lib/Greengrass.jar \
  --trusted-plugin ./GreengrassInstaller/aws.greengrass.crypto.Pkcs11Provider.jar \
  --init-config ./GreengrassInstaller/config.yaml \
  --component-default-user ggc_user:ggc_group \
  --setup-system-service true
```

> **Important**
>
> Windows コアデバイスでは、 を指定--setup-system-service trueして AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります。

--setup-system-service true を指定する場合、ソフトウェアをシステムサービスとしてセットアップして実行したら、インストーラは Successfully set up Nucleus as a system service を出力します。それ以外の場合、正常にソフトウェアがインストールされていれば、インストーラはメッセージを出力しません。

> **Note**
>
> --provision true 引数なしでインストーラを実行するとき、ローカル開発ツールをデプロイするために deploy-dev-tools 引数を使用できません。Greengrass CLI をデバイスに直接デプロイする方法の情報については、「Greengrass コマンドラインインターフェイス」を参照してください。

5. ルートフォルダのファイルを確認して、インストールを確認します。

   Linux or Unix

```
ls /greengrass/v2
```

Windows Command Prompt (CMD)

```
dir C:\greengrass\v2
```

PowerShell

```
ls C:\greengrass\v2
```

インストールが正常に処理された場合、ルートフォルダには `config`、`packages`、`logs` などの複数のフォルダが含まれます。

AWS IoT Greengrass Core ソフトウェアをシステムサービスとしてインストールした場合、インストーラがソフトウェアを実行します。それ以外の場合、ソフトウェアを手動で実行する必要があります。詳細については、「AWS IoT Greengrass Core ソフトウェアを実行する」を参照してください。

ソフトウェアと を設定して使用する方法の詳細については AWS IoT Greengrass、以下を参照してください。

- AWS IoT Greengrass Core ソフトウェアを設定する
- AWS IoT Greengrass コンポーネントを開発する
- デバイスに AWS IoT Greengrass コンポーネントのデプロイ
- Greengrass コマンドラインインターフェイス

AWS IoT フリートプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする

この機能は、Greengrass nucleus コンポーネントの v2.4.0 以降に利用できます。

AWS IoT フリートプロビジョニングを使用すると、X.509 デバイス証明書とプライベートキーが AWS IoT デバイスに初めて接続するときにそれらを生成して安全に配信 AWS IoT するように を設定できます。 は、Amazon ルート認証局 (CA) によって署名されたクライアント証明書 AWS IoT を提供します。フリートプロビジョニングでプロビジョニングする Greengrass コアデバイスのモノのグループ、モノのタイプ、アクセス許可を指定する AWS IoT ように を設定することもできます。プロビジョニングテンプレートを定義して、 が各デバイスを AWS IoT プロビジョニングする方法を定

義します。プロビジョニングテンプレートは、プロビジョニング時にデバイス用に作成するもの、ポリシー、および証明書リソースを指定します。詳細については、「AWS IoT Core デベロッパーガイド」の「プロビジョニングテンプレート」を参照してください。

AWS IoT Greengrass は、 AWS IoT フリートプロビジョニングによって作成された AWS リソースを使用して AWS IoT Greengrass Core ソフトウェアをインストールするために使用できる AWS IoT フリートプロビジョニングプラグインを提供します。フリートプロビジョニングプラグインはクレームによるプロビジョニングを使用します。デバイスは、プロビジョニングクレーム証明書とプライベートキーを使用して、通常の操作に使用できる一意の X.509 デバイス証明書とプライベートキーを取得します。製造中に各デバイスにクレーム証明書とシークレットキーを埋め込むことができるので、お客様は後で各デバイスがオンラインになったときにデバイスをアクティブにできます。複数のデバイスに対して同じクレーム証明書とプライベートキーを使用することができます。詳細については、「AWS IoT Core デベロッパーガイド」の「クレームによるプロビジョニング」を参照してください。

> **Note**
>
> フリートプロビジョニングプラグインは、現在、ハードウェアセキュリティモジュール (HSM) へのプライベートキーと証明書ファイルの保存をサポートしていません。HSM を使用するには、手動プロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールします。

AWS IoT フリートプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールするには、 AWS アカウント が Greengrass コアデバイスのプロビジョニング AWS IoT に使用する にリソースを設定する必要があります。これらのリソースには、プロビジョニングテンプレート、クレーム証明書、およびトークン交換 IAM ロールが含まれます。これらのリソースを作成した後、これらのリソースを再利用して、フリート内の複数のコアデバイスをプロビジョニングできます。詳細については、「Greengrass コアデバイスの AWS IoT フリートプロビジョニングをセットアップする」を参照してください。

> **Important**
>
> AWS IoT Greengrass Core ソフトウェアをダウンロードする前に、コアデバイスが AWS IoT Greengrass Core ソフトウェア v2.0 をインストールして実行するための要件を満たしていることを確認してください。

トピック

前提条件

AWS IoT フリートプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールするには、まず Greengrass コアデバイスの AWS IoT フリートプロビジョニングを設定する必要があります。これらのステップを 1 回完了すると、フリートプロビジョニングを使用して、任意の数のデバイスに AWS IoT Greengrass Core ソフトウェアをインストールできます。

AWS IoT エンドポイントを取得する

の AWS IoT エンドポイントを取得し AWS アカウント、後で使用するために保存します。デバイスはこれらのエンドポイントを使用して AWS IoT に接続します。以下の操作を実行します。

1. AWS IoT のデータエンドポイントを取得します AWS アカウント。

   ```
   aws iot describe-endpoint --endpoint-type iot:Data-ATS
   ```

   要求が正常に処理された場合、レスポンスは次の例のようになります。

   ```
   {
     "endpointAddress": "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
   }
   ```

2. の AWS IoT 認証情報エンドポイントを取得します AWS アカウント。

```
aws iot describe-endpoint --endpoint-type iot:CredentialProvider
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "endpointAddress": "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
}
```

デバイスに証明書をダウンロードする

デバイスは、クレーム証明書とプライベートキーを使用して、 AWS リソースをプロビジョニングし、X.509 デバイス証明書を取得するリクエストを認証します。クレーム証明書とプライベートキーは製造時にデバイスに埋め込むか、 インストール時に証明書とキーをデバイスにコピーすることができます。このセクションでは、クレーム証明書とプライベートキーをデバイスにコピーします。また、デバイスに Amazon ルート認証局 (CA) 証明書もダウンロードします。

> **Important**
>
> プロビジョニングクレームプライベートキーは、Greengrass コアデバイス上にある場合を含め、常に保護する必要があります。Amazon CloudWatch メトリクスとログを使用して、デバイスをプロビジョニングするためのクレーム証明書の不正使用などの誤用の兆候をモニタリングすることをお勧めします。悪用を検出した場合は、プロビジョニングクレーム証明書を無効にして、デバイスのプロビジョニングに使用できないようにします。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT のモニタリング」を参照してください。
>
> に登録するデバイスの数とデバイスを管理しやすくするために AWS アカウント、フリートプロビジョニングテンプレートを作成するときに事前プロビジョニングフックを指定できます。事前プロビジョニングフックは、デバイスが登録時に提供するテンプレートパラメータを検証する AWS Lambda 関数です。例えば、デバイス ID をデータベースと照合して、デバイスにプロビジョニングする権限があることを確認する事前プロビジョニングフックを作成することができます。詳細については、「AWS IoT Core デベロッパーガイド」の「事前プロビジョニングフック」を参照してください。

デバイスにクレーム証明書をダウンロードする

1. クレーム証明書とプライベートキーをデバイスにコピーします。開発用コンピュータとデバイスで SSH と SCP が有効になっている場合、開発用コンピュータで scp コマンドを使用してクレーム証明書とプライベートキーを転送できます。次のコマンド例では、これらのファイルを開発用コンピュータ上の claim-certs という名のフォルダからデバイスに転送します。をデバイスの IP アドレス*device-ip-address*に置き換えます。

   ```
   scp -r claim-certs/ device-ip-address:~
   ```

2. デバイスに Greengrass ルートフォルダを作成します。後で AWS IoT Greengrass Core ソフトウェアをこのフォルダにインストールします。

   Linux or Unix

   - を使用するフォルダ*/greengrass/v2*に置き換えます。

   ```
   sudo mkdir -p /greengrass/v2
   ```

   Windows Command Prompt

   - *C:\greengrass\v2* を使用するフォルダに置き換えます。

   ```
   mkdir C:\greengrass\v2
   ```

   PowerShell

   - *C:\greengrass\v2* を使用するフォルダに置き換えます。

   ```
   mkdir C:\greengrass\v2
   ```

3. (Linux のみ) Greengrass ルートフォルダの親の許可を設定します。

   - */greengrass* をルートフォルダへの親に置き換えます。

   ```
   sudo chmod 755 /greengrass
   ```

4. クレーム証明書を Greengrass ルートフォルダに移動します。

   - */greengrass/v2* または *C:\greengrass\v2* を Greengrass ルートフォルダに置き換えます。

   Linux or Unix

   ```
   sudo mv ~/claim-certs /greengrass/v2
   ```

   Windows Command Prompt (CMD)

   ```
   move %USERPROFILE%\claim-certs C:\greengrass\v2
   ```

   PowerShell

   ```
   mv -Path ~\claim-certs -Destination C:\greengrass\v2
   ```

5. Amazon ルート認証局 (CA) certificate. AWS IoT certificates は、デフォルトで Amazon のルート CA 証明書に関連付けられています。

   Linux or Unix

   ```
   sudo curl -o /greengrass/v2/AmazonRootCA1.pem https://www.amazontrust.com/
   repository/AmazonRootCA1.pem
   ```

   Windows Command Prompt (CMD)

   ```
   curl -o C:\greengrass\v2\\AmazonRootCA1.pem https://www.amazontrust.com/
   repository/AmazonRootCA1.pem
   ```

   PowerShell

   ```
   iwr -Uri https://www.amazontrust.com/repository/AmazonRootCA1.pem -OutFile C:
   \greengrass\v2\\AmazonRootCA1.pem
   ```

デバイス環境をセットアップする

このセクションのステップに従って、 AWS IoT Greengrass コアデバイスとして使用する Linux または Windows デバイスをセットアップします。

Linux デバイスをセットアップする

の Linux デバイスをセットアップするには AWS IoT Greengrass V2

1. AWS IoT Greengrass Core ソフトウェアが実行する必要がある Java ランタイムをインストールします。Amazon Corretto または OpenJDK の長期サポートバージョンを使用することをお勧めします。バージョン 8 以降が必要です。次のコマンドは、デバイスに OpenJDK をインストールする方法を示しています。

   - Debian ベースまたは Ubuntu ベースのディストリビューションの場合:

     ```
     sudo apt install default-jdk
     ```

   - Red Hat ベースのディストリビューションの場合 :

     ```
     sudo yum install java-11-openjdk-devel
     ```

   - 複数 Amazon Linux 2:

     ```
     sudo amazon-linux-extras install java-openjdk11
     ```

   - 複数 Amazon Linux 2023:

     ```
     sudo dnf install java-11-amazon-corretto -y
     ```

   インストールが完了したら、次のコマンドを実行して Java が Linux デバイスで実行されていることを確認します。

   ```
   java -version
   ```

   このコマンドは、デバイス上で実行されている Java のバージョンを出力します。例えば、Debian ベースのディストリビューションでは、出力は次のサンプルのようになります。

   ```
   openjdk version "11.0.9.1" 2020-11-04
   ```

```
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

2. (オプション) デバイスにコンポーネントを実行するデフォルトのシステムユーザーおよびグループを作成します。インストーラ引数`--component-default-user`を使用して、インストール中に AWS IoT Greengrass Core ソフトウェアインストーラにこのユーザーとグループを作成させることもできます。詳細については、「インストーラ引数」を参照してください。

   ```
   sudo useradd --system --create-home ggc_user
   sudo groupadd --system ggc_group
   ```

3. AWS IoT Greengrass Core ソフトウェアを実行するユーザー (通常は `root`) に、任意のユーザーおよび任意のグループ`sudo`で実行するアクセス許可があることを確認します。

   a. `/etc/sudoers` ファイルを開くには、次のコマンドを実行します。

      ```
      sudo visudo
      ```

   b. ユーザーの権限が次の例のようになっていることを確認します。

      ```
      root    ALL=(ALL:ALL) ALL
      ```

4. (オプション) コンテナ化された Lambda 関数を実行するには、cgroups v1 を有効にし、メモリとデバイスの cgroups を有効にしてマウントする必要があります。コンテナ化された Lambda 関数を実行する予定がない場合、この手順を省略できます。

   これらの cgroups オプションを有効にするには、次の Linux カーネルパラメータを使用してデバイスを起動します。

   ```
   cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
   ```

   デバイスのカーネルパラメータを確認および設定するための情報については、オペレーティングシステムおよびブートローダーのドキュメントを参照してください。指示に従って、カーネルパラメータを永続的に設定します。

5. デバイスの要件 にある要件リストで示されているように、その他の必要となる依存関係をすべてデバイスにインストールします。

Windows デバイスをセットアップする

> **Note**
>
> この機能は、Greengrass nucleus コンポーネントの v2.5.0 以降に利用できます。

用の Windows デバイスを設定するには AWS IoT Greengrass V2

1. AWS IoT Greengrass Core ソフトウェアが実行する必要がある Java ランタイムをインストールします。Amazon Corretto または OpenJDK の長期サポートバージョンを使用することをお勧めします。バージョン 8 以降が必要です。
2. PATH システム変数で Java が使用可能か確認し、そうでない場合は追加します。 LocalSystem アカウントは AWS IoT Greengrass Core ソフトウェアを実行するため、ユーザーの PATH ユーザー変数の代わりに PATH システム変数に Java を追加する必要があります。以下の操作を実行します。

   a. Windows キーを押してスタートメニューを開きます。

   b. **`environment variables`** を入力して、スタートメニューからシステムオプションを検索します。

   c. スタートメニューの検索結果から [Edit the system environment variables] (システム環境変数を編集) をクリックして、[System properties] (システムプロパティ) ウィンドウを開きます。

   d. [Environment variables...] (環境変数...) を選択して、[Environment Variables] (環境可変) ウィンドウを開きます。

   e. [System variables] (システム変数) で、[Path] (パス) 、[Edit] (編集) の順に選択します。[Edit environment variable] (環境変数の編集) ウィンドウでは、個別の行に各パスを表示できます。

   f. Java インストールの `bin` フォルダへのパスが存在しているかを確認します。このパスは、次の例のように表示されます。

   ```
   C:\\Program Files\\Amazon Corretto\\jdk11.0.13_8\\bin
   ```

   g. [Path] (パス) で Java インストールの `bin` フォルダが で見つからない場合は、[New] (新規) を選択してこれを追加した上で、[OK] を選択します。

3. 管理者として Windows コマンドプロンプト `cmd.exe` を開きます。

4. Windows デバイスの LocalSystem アカウントにデフォルトユーザーを作成します。*#####*を安全なパスワードに置き換えます。

```
net user /add ggc_user password
```

> **Tip**
>
> Windows の構成によっては、ユーザーのパスワードの期限切れが、将来の日付に設定されている場合があります。Greengrass アプリケーションの動作を継続させるためには、パスワードの有効期限を追跡し、その期限が切れる前に更新します。ユーザーのパスワードには、期限切れを起こさないような設定も可能です。
>
> - ユーザーとパスワードの有効期限を確認するには、次のコマンドを実行します。
>
>   ```
>   net user ggc_user | findstr /C:expires
>   ```
>
> - ユーザーのパスワードが期限切れにならないように設定するには、次のコマンドを実行します。
>
>   ```
>   wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
>   ```
>
> - wmic コマンドが廃止された Windows 10 以降を使用している場合は、次の PowerShell コマンドを実行します。
>
>   ```
>   Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
>    'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
>   ```

5. デバイスに Microsoft から PsExec ユーティリティをダウンロードしてインストールします。
6. PsExec ユーティリティを使用して、デフォルトユーザーのユーザー名とパスワードを LocalSystem アカウントの認証情報マネージャーインスタンスに保存します。*#####*を以前に設定したユーザーのパスワードに置き換えます。

```
psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
```

PsExec License Agreement が開いたら、Accept を選択し、ライセンスに同意してコマンドを実行します。

> **Note**
>
> Windows デバイスでは、 LocalSystem アカウントが Greengrass nucleus を実行するため、 PsExec ユーティリティを使用してデフォルトのユーザー情報を LocalSystem アカウントに格納する必要があります。認証情報マネージャーアプリケーションを使用すると、この情報は アカウントではなく、現在ログオンしているユーザーの Windows LocalSystem アカウントに保存されます。

AWS IoT Greengrass Core ソフトウェアをダウンロードする

AWS IoT Greengrass Core ソフトウェアの最新バージョンは、次の場所からダウンロードできます。

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアの特定のバージョンは、次の場所からダウンロードできます。*#####*をダウンロードするバージョンに置き換えます。
>
> ```
> https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip
> ```

AWS IoT Greengrass Core ソフトウェアをダウンロードするには

1. コアデバイスで、 AWS IoT Greengrass Core ソフトウェアを という名前のファイルにダウンロードしますgreengrass-nucleus-latest.zip。

   Linux or Unix

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
   latest.zip > greengrass-nucleus-latest.zip
   ```

Windows Command Prompt (CMD)

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip > greengrass-nucleus-latest.zip
```

PowerShell

```
iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip -OutFile greengrass-nucleus-latest.zip
```

このソフトウェアをダウンロードすると、Greengrass Core ソフトウェアのライセンス契約に同意したものと見なされます。

2. (オプション) Greengrass nucleus ソフトウェア署名を確認するには

> **Note**
>
> この機能は、Greengrass nucleus バージョン 2.9.5 以降で使用できます。

a. 以下のコマンドを使用して、Greengrass nucleus アーティファクトの署名を確認します。

Linux or Unix

```
jarsigner -verify -certs -verbose greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

インストールする JDK のバージョンによって、ファイル名が異なる場合があります。インストールした JDK のバージョンに *jdk17.0.6_10* を置き換えてください。

```
"C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe" -
verify -certs -verbose greengrass-nucleus-latest.zip
```

PowerShell

インストールする JDK のバージョンによって、ファイル名が異なる場合があります。インストールした JDK のバージョンに *jdk17.0.6_10* を置き換えてください。

```
'C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe' -
verify -certs -verbose greengrass-nucleus-latest.zip
```

b. `jarsigner` が起動すると、検証結果を示す出力が得られます。

   i. Greengrass nucleus の zip ファイルに署名されると、出力に以下のような文が表示されます：

   ```
   jar verified.
   ```

   ii. Greengrass nucleus の zip ファイルに署名されないと、出力に以下のような文が表示されます：

   ```
   jar is unsigned.
   ```

c. Jarsigner `-certs` を `-verify` と `-verbose` オプションと一緒に提供した場合、出力には署名者証明書の詳細情報も含まれます。

3. AWS IoT Greengrass Core ソフトウェアをデバイス上のフォルダに解凍します。を使用するフォルダ*GreengrassInstaller*に置き換えます。

   Linux or Unix

   ```
   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-
   nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir GreengrassInstaller && tar -xf greengrass-nucleus-latest.zip -
   C GreengrassInstaller && del greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   Expand-Archive -Path greengrass-nucleus-latest.zip -DestinationPath .\
   \GreengrassInstaller
   rm greengrass-nucleus-latest.zip
   ```

4. （オプション) 次のコマンドを実行して、 AWS IoT Greengrass Core ソフトウェアのバージョンを確認します。

```
java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
```

> **Important**
>
> v2.4.0 より前のバージョンの Greengrass nucleus をインストールする場合は、 AWS IoT Greengrass Core ソフトウェアをインストールした後にこのフォルダを削除しないでください。 AWS IoT Greengrass Core ソフトウェアは、このフォルダ内のファイルを使用して実行します。
> ソフトウェアの最新バージョンをダウンロードした場合は、v2.4.0 以降をインストールし、AWS IoT Greengrass Core ソフトウェアをインストールした後にこのフォルダを削除できます。

AWS IoT フリートプロビジョニングプラグインをダウンロードする

フリー AWS IoT トプロビジョニングプラグインの最新バージョンは、次の場所からダウンロードできます。

- https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar

> **Note**
>
> フリー AWS IoT トプロビジョニングプラグインの特定のバージョンは、次の場所からダウンロードできます。*#####*をダウンロードするバージョンに置き換えます。フリートプロビジョニングプラグインの各バージョンの詳細については、「AWS IoT フリートプロビジョニングプラグインの変更ログ」を参照してください。
>
> ```
> https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-
> FleetProvisioningByClaim/fleetprovisioningbyclaim-version.jar
> ```

フリートプロビジョニングプラグインはオープンソースです。ソースコードを表示するには、「」のAWS IoT 「 フリートプロビジョニングプラグイン」を参照してください GitHub。

AWS IoT フリートプロビジョニングプラグインをダウンロードするには

- デバイスで、 AWS IoT フリートプロビジョニングプラグインを という名前のファイルにダウンロードしますaws.greengrass.FleetProvisioningByClaim.jar。を使用するフォルダ*GreengrassInstaller*に置き換えます。

  Linux or Unix

  ```
  curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-
  FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar
   > GreengrassInstaller/aws.greengrass.FleetProvisioningByClaim.jar
  ```

  Windows Command Prompt (CMD)

  ```
  curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-
  FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar
   > GreengrassInstaller/aws.greengrass.FleetProvisioningByClaim.jar
  ```

  PowerShell

  ```
  iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-
  FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar -
  OutFile GreengrassInstaller/aws.greengrass.FleetProvisioningByClaim.jar
  ```

  このソフトウェアをダウンロードすると、Greengrass Core ソフトウェアのライセンス契約に同意したものと見なされます。

AWS IoT Greengrass Core ソフトウェアのインストール

次のアクションを指定する引数を含むインストーラを実行します。

- フリートプロビジョニングプラグインを使用して AWS リソースをプロビジョニングするように指定する部分設定ファイルからインストールします。 AWS IoT Greengrass Core ソフトウェアは、デバイス上のすべての Greengrass コンポーネントの設定を指定する設定ファイルを使用します。インストーラは、提供される部分設定ファイルと、フリートプロビジョニングプラグインが作成する AWS リソースから、完全な設定ファイルを作成します。
- コアデバイスでソフトウェアコンポーネントを実行するために ggc_user システムユーザーを使用するように指定します。Linux デバイスでは、このコマンドも ggc_group システムグループを

使用するように指定し、さらにインストーラによってシステムユーザーとグループが、ユーザーに代わって作成されます。

- 起動時に実行されるシステムサービスとして AWS IoT Greengrass Core ソフトウェアを設定します。Linux デバイスでは、これは Systemd init システムが必要です。

> **Important**
>
> Windows コアデバイスでは、 AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります。

指定できる引数の詳細については、「インストーラ引数」を参照してください。

> **Note**
>
> メモリが制限された AWS IoT Greengrass デバイスで実行している場合は、 AWS IoT Greengrass Core ソフトウェアが使用するメモリ量を制御できます。メモリ割り当てを制御するには、nucleus コンポーネントの `jvmOptions` 設定パラメータで JVM ヒープのサイズオプションを設定できます。詳細については、「JVM オプションでメモリ割り当てを制御する」を参照してください。

AWS IoT Greengrass Core ソフトウェアをインストールするには

1. AWS IoT Greengrass Core ソフトウェアのバージョンを確認します。

   - を、ソフトウェアを含むフォルダへのパス*GreengrassInstaller*に置き換えます。

   ```
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

2. テキストエディタを使用し、`config.yaml` という名前の設定ファイルを作成してインストーラに提供します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

   ```
   nano GreengrassInstaller/config.yaml
   ```

次の YAML コンテンツをファイルにコピーします。この部分設定ファイルは、フリートプロビジョニングプラグインのパラメータを指定します。指定できるオプションの詳細については、「AWS IoT フリートプロビジョニングプラグインを設定する」を参照してください。

Linux or Unix

```
---
services:
  aws.greengrass.Nucleus:
    version: "2.12.2"
  aws.greengrass.FleetProvisioningByClaim:
    configuration:
      rootPath: "/greengrass/v2"
      awsRegion: "us-west-2"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      iotCredentialEndpoint: "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      provisioningTemplate: "GreengrassFleetProvisioningTemplate"
      claimCertificatePath: "/greengrass/v2/claim-certs/claim.pem.crt"
      claimCertificatePrivateKeyPath: "/greengrass/v2/claim-certs/
claim.private.pem.key"
      rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
      templateParameters:
        ThingName: "MyGreengrassCore"
        ThingGroupName: "MyGreengrassCoreGroup"
```

Windows

```
---
services:
  aws.greengrass.Nucleus:
    version: "2.12.2"
  aws.greengrass.FleetProvisioningByClaim:
    configuration:
      rootPath: "C:\\greengrass\\v2"
      awsRegion: "us-west-2"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      iotCredentialEndpoint: "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      provisioningTemplate: "GreengrassFleetProvisioningTemplate"
```

```
      claimCertificatePath: "C:\\greengrass\\v2\\claim-certs\\claim.pem.crt"
      claimCertificatePrivateKeyPath: "C:\\greengrass\\v2\\claim-certs\
\claim.private.pem.key"
      rootCaPath: "C:\\greengrass\\v2\\AmazonRootCA1.pem"
      templateParameters:
        ThingName: "MyGreengrassCore"
        ThingGroupName: "MyGreengrassCoreGroup"
```

次に、以下の操作を実行します。

- *2.12.2* を AWS IoT Greengrass Core ソフトウェアのバージョンに置き換えます。
- */greengrass/v2* または *C:\greengrass\v2* の各インスタンスを Greengrass ルートフォルダに置き換えます。

  > **Note**
  >
  > Windows デバイスでは、パスセパレータは二重バックスラッシュ (\\) で指定する必要があります (例: `C:\\greengrass\\v2`)。

- *us-west-2* を、プロビジョニングテンプレートやその他のリソースを作成した AWS リージョンに置き換えます。
- を AWS IoT データエンドポイント`iotDataEndpoint`に置き換えます。
- を AWS IoT 認証情報エンドポイント`iotCredentialEndpoint`に置き換えます。
- をトークン交換ロールエイリアスの名前*GreengrassCoreTokenExchangeRoleAlias*に置き換えます。
- をフリートプロビジョニングテンプレートの名前*GreengrassFleetProvisioningTemplate*に置き換えます。
- `claimCertificatePath` をデバイス上のクレーム証明書へのパスに置き換えます。
- `claimCertificatePrivateKeyPath` をデバイス上のクレーム証明書のプライベートキーへのパスに置き換えます。
- テンプレートパラメータ (`templateParameters`) を、デバイスのプロビジョニングに使用する値に置き換えます。この例は、`ThingName` および `ThingGroupName` パラメータを定義するテンプレート例を参照しています。

Note

この設定ファイルでは、使用するポートやネットワークプロキシなど、次の例で示すように、他の設定オプションをカスタマイズすることができます。詳細については、Greengrass nucleus 設定を参照してください。

Linux or Unix

```
---
services:
  aws.greengrass.Nucleus:
    version: "2.12.2"
    configuration:
      mqtt:
        port: 443
      greengrassDataPlanePort: 443
      networkProxy:
        noProxyAddresses: "http://192.168.0.1,www.example.com"
        proxy:
          url: "http://my-proxy-server:1100"
          username: "Mary_Major"
          password: "pass@word1357"
  aws.greengrass.FleetProvisioningByClaim:
    configuration:
      rootPath: "/greengrass/v2"
      awsRegion: "us-west-2"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-
west-2.amazonaws.com"
      iotCredentialEndpoint: "device-credentials-
prefix.credentials.iot.us-west-2.amazonaws.com"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      provisioningTemplate: "GreengrassFleetProvisioningTemplate"
      claimCertificatePath: "/greengrass/v2/claim-certs/claim.pem.crt"
      claimCertificatePrivateKeyPath: "/greengrass/v2/claim-certs/
claim.private.pem.key"
      rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
      templateParameters:
        ThingName: "MyGreengrassCore"
        ThingGroupName: "MyGreengrassCoreGroup"
      mqttPort: 443
      proxyUrl: "http://my-proxy-server:1100"
```

```
      proxyUserName: "Mary_Major"
      proxyPassword: "pass@word1357"
```

Windows

```
---
services:
  aws.greengrass.Nucleus:
    version: "2.12.2"
    configuration:
      mqtt:
        port: 443
      greengrassDataPlanePort: 443
      networkProxy:
        noProxyAddresses: "http://192.168.0.1,www.example.com"
        proxy:
          url: "http://my-proxy-server:1100"
          username: "Mary_Major"
          password: "pass@word1357"
  aws.greengrass.FleetProvisioningByClaim:
    configuration:
      rootPath: "C:\\greengrass\\v2"
      awsRegion: "us-west-2"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-
west-2.amazonaws.com"
      iotCredentialEndpoint: "device-credentials-
prefix.credentials.iot.us-west-2.amazonaws.com"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      provisioningTemplate: "GreengrassFleetProvisioningTemplate"
      claimCertificatePath: "C:\\greengrass\\v2\\claim-certs\
\claim.pem.crt"
      claimCertificatePrivateKeyPath: "C:\\greengrass\\v2\\claim-certs\
\claim.private.pem.key"
      rootCaPath: "C:\\greengrass\\v2\\AmazonRootCA1.pem"
      templateParameters:
        ThingName: "MyGreengrassCore"
        ThingGroupName: "MyGreengrassCoreGroup"
      mqttPort: 443
      proxyUrl: "http://my-proxy-server:1100"
      proxyUserName: "Mary_Major"
      proxyPassword: "pass@word1357"
```

HTTPS プロキシを使用するには、フリートプロビジョニングプラグインのバージョン 1.1.0 以降を使用する必要があります。次の例に示すように、さらに、`system` にある `rootCaPath` も指定する必要があります。

Linux or Unix

```
---
system:
  rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
services:
  ...
```

Windows

```
---
system:
  rootCaPath: "C:\\greengrass\\v2\\AmazonRootCA1.pem"
services:
  ...
```

3. インストーラーを実行します。`--trusted-plugin` を指定してフリートプロビジョニングプラグインを提供し、`--init-config` を指定して設定ファイルを提供します。

   - を Greengrass ルートフォルダ */greengrass/v2* に置き換えます。
   - の各インスタンスを、インストーラを解凍したフォルダ *GreengrassInstaller* に置き換えます。

   Linux or Unix

   ```
   sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
     -jar ./GreengrassInstaller/lib/Greengrass.jar \
     --trusted-plugin ./GreengrassInstaller/
   aws.greengrass.FleetProvisioningByClaim.jar \
     --init-config ./GreengrassInstaller/config.yaml \
     --component-default-user ggc_user:ggc_group \
     --setup-system-service true
   ```

Windows Command Prompt (CMD)

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" ^
  -jar ./GreengrassInstaller/lib/Greengrass.jar ^
  --trusted-plugin ./GreengrassInstaller/
aws.greengrass.FleetProvisioningByClaim.jar ^
  --init-config ./GreengrassInstaller/config.yaml ^
  --component-default-user ggc_user ^
  --setup-system-service true
```

PowerShell

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" `
  -jar ./GreengrassInstaller/lib/Greengrass.jar `
  --trusted-plugin ./GreengrassInstaller/
aws.greengrass.FleetProvisioningByClaim.jar `
  --init-config ./GreengrassInstaller/config.yaml `
  --component-default-user ggc_user `
  --setup-system-service true
```

Important

Windows コアデバイスでは、 を指定`--setup-system-service true`して AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります。

`--setup-system-service true` を指定する場合、ソフトウェアをシステムサービスとしてセットアップして実行したら、インストーラは `Successfully set up Nucleus as a system service` を出力します。それ以外の場合、正常にソフトウェアがインストールされていれば、インストーラはメッセージを出力しません。

Note

`--provision true` 引数なしでインストーラを実行するとき、ローカル開発ツールをデプロイするために `deploy-dev-tools` 引数を使用できません。Greengrass CLI をデバイスに直接デプロイする方法の情報については、「Greengrass コマンドラインインターフェイス」を参照してください。

4. ルートフォルダのファイルを確認して、インストールを確認します。

   Linux or Unix

   ```
   ls /greengrass/v2
   ```

   Windows Command Prompt (CMD)

   ```
   dir C:\greengrass\v2
   ```

   PowerShell

   ```
   ls C:\greengrass\v2
   ```

   インストールが正常に処理された場合、ルートフォルダには `config`、`packages`、`logs` などの複数のフォルダが含まれます。

AWS IoT Greengrass Core ソフトウェアをシステムサービスとしてインストールした場合、インストーラがソフトウェアを実行します。それ以外の場合、ソフトウェアを手動で実行する必要があります。詳細については、「AWS IoT Greengrass Core ソフトウェアを実行する」を参照してください。

ソフトウェアと を設定して使用する方法の詳細については AWS IoT Greengrass、以下を参照してください。

- AWS IoT Greengrass Core ソフトウェアを設定する
- AWS IoT Greengrass コンポーネントを開発する
- デバイスに AWS IoT Greengrass コンポーネントのデプロイ
- Greengrass コマンドラインインターフェイス

Greengrass コアデバイスの AWS IoT フリートプロビジョニングをセットアップする

AWS IoT Greengrass Core ソフトウェアをフリートプロビジョニングでインストールするには、まず、次のリソースを AWS アカウント でセットアップする必要があります。これらのリソースにより、デバイスは自身を AWS IoT に登録することができるようになり、Greengrass コアデバイスと

して動作します。このセクションの手順を一度実行して、これらのリソースを AWS アカウント で設定してください。

- トークン交換 IAM ロールは、コアデバイスが AWS サービスへの呼び出しを認証するために使用します。
- AWS IoT ロールエイリアスは、トークン交換ロールを指します。
- (オプション) AWS IoT ポリシーは、コアデバイスが AWS IoT および AWS IoT Greengrass サービスへの呼び出しを認証するために使用します。この AWS IoT ポリシーは、トークン交換ロールを指している AWS IoT ロールエイリアスの `iot:AssumeRoleWithCertificate` 許可を許可する必要があります。

  単一の AWS IoT ポリシーをフリート内のすべてのコアデバイスに使用するか、フリートプロビジョニングテンプレートを設定して、各コアデバイスに対して AWS IoT ポリシーを作成することができます。
- AWS IoT フリートプロビジョニングテンプレート。このテンプレートでは、以下を指定する必要があります。
  - AWS IoT モノのリソース。既存のモノのグループのリストを指定して、オンラインになったときに各デバイスにコンポーネントをデプロイできます。
  - AWS IoT ポリシーリソース。このリソースは、次のいずれかのプロパティを定義できます。
    - 既存の AWS IoT ポリシーの名前。このオプションを選択した場合、このテンプレートから作成されたコアデバイスは同じ AWS IoT ポリシーを使用するようになり、そのアクセス許可はフリートとして管理することができます。
    - AWS IoT ポリシードキュメント。このオプションを選択した場合、このテンプレートから作成する各コアデバイスは、一意の AWS IoT ポリシーを使用するようになり、各コアデバイスのアクセス許可を管理することができます。
  - AWS IoT 証明書リソース。この証明書リソースでは、`AWS::IoT::Certificate::Id` パラメータを使用して、証明書をコアデバイスにアタッチする必要があります。詳細については、「AWS IoTデベロッパーガイド」の「J ust-in-time プロビジョニング」を参照してください。
- フリートプロビジョニングテンプレートの AWS IoT プロビジョニングクレーム証明書とプライベートキーです。製造中にこの証明書とプライベートキーをデバイスに埋め込むことができます。埋め込むと、デバイスはオンライン状態になったときにデバイスを登録してプロビジョニングすることができます。

> **Important**
>
> プロビジョニングクレームプライベートキーは、Greengrass コアデバイス上にある場合を含め、常に保護する必要があります。Amazon CloudWatch メトリクスとログを使用して、デバイスをプロビジョニングするためのクレーム証明書の不正使用などの誤用の兆候をモニタリングすることをお勧めします。悪用を検出した場合は、プロビジョニングクレーム証明書を無効にして、デバイスのプロビジョニングに使用できないようにします。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT のモニタリング」を参照してください。
>
> デバイス数と AWS アカウント に自身を登録するデバイスについてより適切に管理できるように、フリートプロビジョニングテンプレートを作成するときに、事前プロビジョニングフックを指定することができます。事前プロビジョニングフックは、登録時にデバイスが提供するテンプレートパラメータを検証する AWS Lambda 関数です。例えば、デバイス ID をデータベースと照合して、デバイスにプロビジョニングする権限があることを確認する事前プロビジョニングフックを作成することができます。詳細については、「AWS IoT Core デベロッパーガイド」の「事前プロビジョニングフック」を参照してください。

- プロビジョニングクレーム証明書にアタッチして、デバイスがフリートプロビジョニングテンプレートを登録および使用できるようにする AWS IoT ポリシー。

トピック

- トークン交換ロールを作成する
- AWS IoT ポリシーを作成する
- フリートプロビジョニングテンプレートを作成する
- プロビジョニングクレーム証明書とプライベートキーを作成する

トークン交換ロールを作成する

Greengrass コアデバイスは、トークン交換ロールという IAM サービスロールを使用して、AWS サービスへの通話を承認します。デバイスはAWS IoT認証情報プロバイダーを使用して、このロールの一時的なAWS認証情報を取得します。これにより、デバイスは とやり取りしAWS IoT、Amazon CloudWatch Logs にログを送信し、Amazon S3 からカスタムコンポーネントアーティファクトをダウンロードできるようになります。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

AWS IoT のロールエイリアスを使用して、Greengrass コアデバイスのトークン交換ロールを設定します。ロールエイリアスは、デバイスのトークン交換ロールを変更できるようにしますが、デバイス設定は同じ内容に保たれます。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS サービスへの直接呼び出しを認証する」を参照してください。

このセクションでは、トークン交換 IAM ロールとロールを指す AWS IoT ロールエイリアスを作成します。Greengrass コアデバイスを既に設定している場合、新しく作成せず、トークン交換ロールとロールエイリアスを使用できます。

トークン交換 IAM ロールを作成するには

1. デバイスがトークン交換ロールとして使用できる IAM ロールを作成します。以下の操作を実行します。

   a. トークン交換ロールが必要とする、信頼できるポリシードキュメントが含まれるファイルを作成します。

      例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

      ```
      nano device-role-trust-policy.json
      ```

      次の JSON をファイルにコピーします。

      ```
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Principal": {
              "Service": "credentials.iot.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
          }
        ]
      }
      ```

   b. 信頼ポリシードキュメントでトークン交換ロールを作成します。

      - *GreengrassV2TokenExchangeRole* を作成する IAM ロールの名前に置き換えます。

```
aws iam create-role --role-name GreengrassV2TokenExchangeRole --assume-role-
policy-document file://device-role-trust-policy.json
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "Role": {
    "Path": "/",
    "RoleName": "GreengrassV2TokenExchangeRole",
    "RoleId": "AROAZ2YMUHYHK5OKM77FB",
    "Arn": "arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole",
    "CreateDate": "2021-02-06T00:13:29+00:00",
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "credentials.iot.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
  }
```

c. トークン交換ロールが必要なアクセスポリシードキュメントを含むファイルを作成します。

例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano device-role-access-policy.json
```

次の JSON をファイルにコピーします。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
```

```
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
}
```

> **Note**
>
> このアクセスポリシーでは、S3 バケットのコンポーネントアーティファクトへのアクセスが許可されていません。Amazon S3 でアーティファクトを定義するカスタムコンポーネントをデプロイするには、コアデバイスがコンポーネントアーティファクトを取得できるようにする許可をロールに追加する必要があります。詳細については、「コンポーネントのアーティファクトの S3 バケットへのアクセスを許可する」を参照してください。
> コンポーネントアーティファクトに S3 バケットをまだ持っていない場合、バケットを作成した後でこれらのアクセス許可を追加できます。

d. ポリシードキュメントから IAM ポリシーを作成します。

- *GreengrassV2TokenExchangeRoleAccess* を作成する IAM ポリシーの名前に置き換えます。

```
aws iam create-policy --policy-name GreengrassV2TokenExchangeRoleAccess --
policy-document file://device-role-access-policy.json
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "Policy": {
    "PolicyName": "GreengrassV2TokenExchangeRoleAccess",
    "PolicyId": "ANPAZ2YMUHYHACI7C5Z66",
    "Arn": "arn:aws:iam::123456789012:policy/
GreengrassV2TokenExchangeRoleAccess",
```

```
    "Path": "/",
    "DefaultVersionId": "v1",
    "AttachmentCount": 0,
    "PermissionsBoundaryUsageCount": 0,
    "IsAttachable": true,
    "CreateDate": "2021-02-06T00:37:17+00:00",
    "UpdateDate": "2021-02-06T00:37:17+00:00"
  }
}
```

e. IAM ポリシーをトークン交換ロールにアタッチします。

- *GreengrassV2TokenExchangeRole* を IAM ロールの名前に置き換えます。
- ポリシー ARN を前のステップで作成した IAM ポリシーの ARN に置き換えます。

```
aws iam attach-role-policy --role-name GreengrassV2TokenExchangeRole --policy-
arn arn:aws:iam::123456789012:policy/GreengrassV2TokenExchangeRoleAccess
```

要求が正常に処理された場合、コマンドは出力されません。

2. トークン交換ロールを指す AWS IoT ロールエイリアスを作成します。

- を、作成するロールエイリアスの名前*GreengrassCoreTokenExchangeRoleAlias*に置き換えます。
- ロール ARN を前のステップで作成した IAM ロールの ARN に置き換えます。

```
aws iot create-role-alias --role-alias GreengrassCoreTokenExchangeRoleAlias --role-
arn arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "roleAlias": "GreengrassCoreTokenExchangeRoleAlias",
  "roleAliasArn": "arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias"
}
```

> **Note**
>
> ロールエイリアスを作成するには、トークン交換 IAM ロールを AWS IoT に渡す許可が必要です。ロールエイリアスの作成時にエラーメッセージが表示された場合、AWS ユーザーがこの許可を得ていることを確認してください。詳細については、「AWS Identity and Access Management ユーザーガイド」の「AWS サービスにロールをわたす許可をユーザーに付与する」を参照してください。

AWS IoT ポリシーを作成する

デバイスを AWS IoT モノとして登録した後、そのデバイスはデジタル証明書を使用して AWS に認証できます。この証明書には、デバイスが証明書で使用できるアクセス許可を定義する 1 つ以上の AWS IoT ポリシーが含まれています。これらのポリシーにより、デバイスはと AWS IoT および AWS IoT Greengrass と通信できるようになります。

AWS IoT フリートプロビジョニングにより、デバイスは AWS IoT に接続し、デバイス証明書を作成してダウンロードします。次のセクションで作成するフリートプロビジョニングテンプレートでは、AWS IoT がすべてのデバイスの証明書に同じ AWS IoT ポリシーを添付するか、各デバイス向けに新しいポリシーを作成するかを指定できます。

このセクションでは、AWS IoT がすべてのデバイスの証明書にアタッチする AWS IoT ポリシーを作成します。この方法では、すべてのデバイスの権限をフリートとして管理できます。代わりに、各デバイスに対して新しい AWS IoT ポリシーを作成するのであれば、このセクションをスキップして、フリートテンプレートを定義するときにそのポリシーを参照することができます。

AWS IoT ポリシーを作成するには

- Greengrass コアデバイスのフリートへの AWS IoT 権限を定義する AWS IoT ポリシーを作成します。次のポリシーは、すべての MQTT トピックと Greengrass 操作へのアクセスを許可するため、デバイスがカスタムアプリケーションや新しい Greengrass 操作を必要とする今後の変更でも動作するようになります。このポリシーは `iot:AssumeRoleWithCertificate` アクセス許可を付与するため、デバイスで前のセクションで作成したトークン交換ロールを使用できるようになります。ユースケースに基づいてこのポリシーを制限できます。詳細については、「AWS IoT Greengrass V2 コアデバイス向けの最低限の AWS IoT ポリシー」を参照してください。

  以下の操作を実行します。

a. Greengrass コアデバイスが必要な AWS IoT ポリシードキュメントを含むファイルを作成します。

例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano greengrass-v2-iot-policy.json
```

次の JSON をファイルにコピーします。

- `iot:AssumeRoleWithCertificate` リソースを、前のセクションで作成した AWS IoT ロールエイリアスの ARN に置き換えます。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Subscribe",
        "iot:Receive",
        "iot:Connect",
        "greengrass:*"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iot:AssumeRoleWithCertificate",
      "Resource": "arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias"
    }
  ]
}
```

b. ポリシードキュメントから AWS IoT ポリシーを作成します。

- *GreengrassV2IoTThingPolicy* を、作成するポリシーの名前に置き換えます。

```
aws iot create-policy --policy-name GreengrassV2IoTThingPolicy --policy-
document file://greengrass-v2-iot-policy.json
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "policyName": "GreengrassV2IoTThingPolicy",
  "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy",
  "policyDocument": "{
    \"Version\": \"2012-10-17\",
    \"Statement\": [
      {
        \"Effect\": \"Allow\",
        \"Action\": [
          \"iot:Publish\",
          \"iot:Subscribe\",
          \"iot:Receive\",
          \"iot:Connect\",
          \"greengrass:*\"
        ],
        \"Resource\": [
          \"*\"
        ]
      },
      {
        \"Effect\": \"Allow\",
        \"Action\": \"iot:AssumeRoleWithCertificate\",
        \"Resource\": \"arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias\"
      }
    ]
  }",
  "policyVersionId": "1"
}
```

フリートプロビジョニングテンプレートを作成する

AWS IoT フリートプロビジョニングテンプレートは、AWS IoT モノ、ポリシーおよび証明書をプロビジョニングする方法を定義します。フリートプロビジョニングプラグインを使用して Greengrass コアデバイスをプロビジョニングするには、次の事項を指定するテンプレートを作成する必要があります。

- AWS IoT モノのリソース。既存のモノのグループのリストを指定して、オンラインになったときに各デバイスにコンポーネントをデプロイできます。
- AWS IoT ポリシーリソース。このリソースは、次のいずれかのプロパティを定義できます。
  - 既存の AWS IoT ポリシーの名前。このオプションを選択した場合、このテンプレートから作成されたコアデバイスは同じ AWS IoT ポリシーを使用するようになり、そのアクセス許可はフリートとして管理することができます。
  - AWS IoT ポリシードキュメント。このオプションを選択した場合、このテンプレートから作成する各コアデバイスは、一意の AWS IoT ポリシーを使用するようになり、各コアデバイスのアクセス許可を管理することができます。
- AWS IoT 証明書リソース。この証明書リソースでは、`AWS::IoT::Certificate::Id` パラメータを使用して、証明書をコアデバイスにアタッチする必要があります。詳細については、「 AWS IoTデベロッパーガイド」の「J ust-in-time プロビジョニング」を参照してください。

テンプレートでは、既存のモノグループのリストに AWS IoT モノを追加するように指定できます。コアデバイスが初めて AWS IoT Greengrass に接続するときに、メンバーになっているモノグループごとに Greengrass のデプロイを受信します。モノグループを使用して、最新のソフトウェアがオンラインになり次第、各デバイスにデプロイすることができます。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

AWS IoT サービスでは、デバイスのプロビジョニング時に、AWS アカウント に AWS IoT リソースを作成し、更新する権限が必要です。AWS IoT サービスアクセスを許可するには、IAM ロールを作成し、テンプレートの作成時に提供します。 AWS IoTにはAWSIoTThingsRegistration、デバイスのプロビジョニング時に がAWS IoT使用するすべてのアクセス許可へのアクセスを許可する マネージドポリシー が用意されています。このマネージドポリシーを使用するか、ユースケースのマネージドポリシーのアクセス許可に範囲を絞り込んだカスタムポリシーを作成することができます。

このセクションでは、AWS IoT がデバイスのリソースをプロビジョニングできるようにする IAM ロールを作成し、その IAM ロールを使用するフリートプロビジョニングテンプレートを作成します。

フリートプロビジョニングテンプレートを作成するには

1. AWS アカウント にリソースをプロビジョニングするために、AWS IoT が継承することができる IAM ロールを作成します。以下の操作を実行します。

   a. AWS IoT がロールを継承できるようにするための、信頼できるポリシードキュメントが含まれるファイルを作成します。

      例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

      ```
      nano aws-iot-trust-policy.json
      ```

      次の JSON をファイルにコピーします。

      ```
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Principal": {
              "Service": "iot.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
          }
        ]
      }
      ```

   b. 信頼できるポリシードキュメントを使用して IAM ロールを作成します。

      - を、作成する IAM ロールの名前*GreengrassFleetProvisioningRole*に置き換えます。

      ```
      aws iam create-role --role-name GreengrassFleetProvisioningRole --assume-role-
      policy-document file://aws-iot-trust-policy.json
      ```

      要求が正常に処理された場合、レスポンスは次の例のようになります。

      ```
      {
        "Role": {
      ```

```
    "Path": "/",
    "RoleName": "GreengrassFleetProvisioningRole",
    "RoleId": "AROAZ2YMUHYHK5OKM77FB",
    "Arn": "arn:aws:iam::123456789012:role/GreengrassFleetProvisioningRole",
    "CreateDate": "2021-07-26T00:15:12+00:00",
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "iot.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
  }
}
```

c. AWSIoTThingsRegistration ポリシーを確認します。これにより、デバイスのプロビジョニング時に が使用するAWS IoT可能性のあるすべてのアクセス許可へのアクセスが許可されます。このマネージドポリシーを使用するか、ユースケースのアクセス許可に範囲を絞り込んだカスタムポリシーを作成することができます。カスタムポリシーを作成する場合は、ここで作成します。

d. IAM ポリシーをフリートプロビジョニングロールにアタッチします。

- *GreengrassFleetProvisioningRole* を IAM ロールの名前に置き換えます。
- 前のステップでカスタムポリシーを作成した場合は、ポリシー ARN を使用する IAM ポリシーの ARN に置き換えます。

```
aws iam attach-role-policy --role-name GreengrassFleetProvisioningRole --
policy-arn arn:aws:iam::aws:policy/service-role/AWSIoTThingsRegistration
```

要求が正常に処理された場合、コマンドは出力されません。

2. (オプション) 事前プロビジョニングフックを作成します。これは、デバイスが登録時に提供するテンプレートパラメータを検証する AWS Lambda 関数です。事前プロビジョニングフックを使用することで、AWS アカウント アカウントに搭載するデバイスの種類と台数をより細かく制御

することができます。詳細については、「AWS IoT Core デベロッパーガイド」の「事前プロビジョニングフック」を参照してください。

3. フリートプロビジョニングテンプレートを作成します。以下の操作を実行します。

   a. プロビジョニングテンプレートドキュメントを含めるファイルを作成します。

      例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

      ```
      nano greengrass-fleet-provisioning-template.json
      ```

      プロビジョニングテンプレートドキュメントを記述します。下記のプロビジョニングテンプレート例から始めることができます。この例では、以下のプロパティを持つ AWS IoT モノを作成するように指定しています。

      - モノの名前は、ThingName テンプレートパラメータで指定した値です。
      - モノは、ThingGroupName テンプレートパラメータで指定したモノグループのメンバーです。モノグループは、AWS アカウント に存在する必要があります。
      - モノの証明書には、GreengrassV2IoTThingPolicy と名前が付けられた AWS IoT ポリシーがアタッチされています。

      詳細については、「AWS IoT Core デベロッパーガイド」の「プロビジョニングテンプレート」を参照してください。

      ```
      {
        "Parameters": {
          "ThingName": {
            "Type": "String"
          },
          "ThingGroupName": {
            "Type": "String"
          },
          "AWS::IoT::Certificate::Id": {
            "Type": "String"
          }
        },
        "Resources": {
          "MyThing": {
            "OverrideSettings": {
      ```

```
        "AttributePayload": "REPLACE",
        "ThingGroups": "REPLACE",
        "ThingTypeName": "REPLACE"
      },
      "Properties": {
        "AttributePayload": {},
        "ThingGroups": [
          {
            "Ref": "ThingGroupName"
          }
        ],
        "ThingName": {
          "Ref": "ThingName"
        }
      },
      "Type": "AWS::IoT::Thing"
    },
    "MyPolicy": {
      "Properties": {
        "PolicyName": "GreengrassV2IoTThingPolicy"
      },
      "Type": "AWS::IoT::Policy"
    },
    "MyCertificate": {
      "Properties": {
        "CertificateId": {
          "Ref": "AWS::IoT::Certificate::Id"
        },
        "Status": "Active"
      },
      "Type": "AWS::IoT::Certificate"
    }
  }
}
```

> **Note**
>
> *MyThing*、*MyPolicy*、*MyCertificate*は、フリートプロビジョニングテンプレート内の各リソース仕様を識別する任意の名前です。 AWS IoTは、テンプレートから作成するリソースでこれらの名前を使用しません。これらの名前を使用する

か、テンプレート内の各リソースを識別するのに役立つ値に置き換えることができます。

b. プロビジョニングテンプレートドキュメントからフリートプロビジョニングテンプレートを作成します。

- を、作成するテンプレートの名前*GreengrassFleetProvisioningTemplate*に置き換えます。
- テンプレートの説明を作成するテンプレートの説明に置き換えます。
- プロビジョニングロールの ARN を、先程作成したロールの ARN に置き換えます。

Linux or Unix

```
aws iot create-provisioning-template \
  --template-name GreengrassFleetProvisioningTemplate \
  --description "A provisioning template for Greengrass core devices." \
  --provisioning-role-arn "arn:aws:iam::123456789012:role/
GreengrassFleetProvisioningRole" \
  --template-body file://greengrass-fleet-provisioning-template.json \
  --enabled
```

Windows Command Prompt (CMD)

```
aws iot create-provisioning-template ^
  --template-name GreengrassFleetProvisioningTemplate ^
  --description "A provisioning template for Greengrass core devices." ^
  --provisioning-role-arn "arn:aws:iam::123456789012:role/
GreengrassFleetProvisioningRole" ^
  --template-body file://greengrass-fleet-provisioning-template.json ^
  --enabled
```

PowerShell

```
aws iot create-provisioning-template `
  --template-name GreengrassFleetProvisioningTemplate `
  --description "A provisioning template for Greengrass core devices." `
  --provisioning-role-arn "arn:aws:iam::123456789012:role/
GreengrassFleetProvisioningRole" `
  --template-body file://greengrass-fleet-provisioning-template.json `
```

```
    --enabled
```

> **Note**
>
> 事前プロビジョニングフックを作成した場合は、事前プロビジョニングフックの Lambda 関数の ARN を `--pre-provisioning-hook` 引数で指定します。
>
> ```
> --pre-provisioning-hook targetArn=arn:aws:lambda:us-
> west-2:123456789012:function:GreengrassPreProvisioningHook
> ```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
    "templateArn": "arn:aws:iot:us-west-2:123456789012:provisioningtemplate/
GreengrassFleetProvisioningTemplate",
    "templateName": "GreengrassFleetProvisioningTemplate",
    "defaultVersionId": 1
}
```

プロビジョニングクレーム証明書とプライベートキーを作成する

クレーム証明書は、デバイスを AWS IoT モノとして登録し、通常の操作に使用するための固有の X.509 デバイス証明書を取得するための X.509証明書 です。クレーム証明書を作成した後、デバイスが使用できる AWS IoT ポリシーをアタッチして、固有のデバイス証明書を作成し、フリートプロビジョニングテンプレートを使用してプロビジョニングします。クレーム証明書を持つデバイスは、AWS IoT ポリシーで許可したプロビジョニングテンプレートを使用してのみプロビジョニングできます。

このセクションでは、クレーム証明書を作成し、前のセクションで作成したフリートプロビジョニングテンプレートで使用するデバイス用に設定します。

> **Important**
>
> プロビジョニングクレームプライベートキーは、Greengrass コアデバイス上にある場合を含め、常に保護する必要があります。Amazon CloudWatch メトリクスとログを使用して、デバイスをプロビジョニングするためのクレーム証明書の不正使用などの誤用の兆候をモニ

タリングすることをお勧めします。悪用を検出した場合は、プロビジョニングクレーム証明書を無効にして、デバイスのプロビジョニングに使用できないようにします。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT のモニタリング」を参照してください。
デバイス数と AWS アカウント に自身を登録するデバイスについてより適切に管理できるように、フリートプロビジョニングテンプレートを作成するときに、事前プロビジョニングフックを指定することができます。事前プロビジョニングフックは、登録時にデバイスが提供するテンプレートパラメータを検証する AWS Lambda 関数です。例えば、デバイス ID をデータベースと照合して、デバイスにプロビジョニングする権限があることを確認する事前プロビジョニングフックを作成することができます。詳細については、「AWS IoT Core デベロッパーガイド」の「事前プロビジョニングフック」を参照してください。

プロビジョニングクレーム証明書とプライベートキーを作成するには

1. クレーム証明書とプライベートキーをダウンロードするフォルダを作成します。

```
mkdir claim-certs
```

2. プロビジョニングに使用する証明書とプライベートキーを作成します。AWS IoT は、Amazon ルート認証局 (CA) によって署名されたクライアント証明書を提供します。

   Linux or Unix

```
aws iot create-keys-and-certificate \
  --certificate-pem-outfile "claim-certs/claim.pem.crt" \
  --public-key-outfile "claim-certs/claim.public.pem.key" \
  --private-key-outfile "claim-certs/claim.private.pem.key" \
  --set-as-active
```

   Windows Command Prompt (CMD)

```
aws iot create-keys-and-certificate ^
  --certificate-pem-outfile "claim-certs/claim.pem.crt" ^
  --public-key-outfile "claim-certs/claim.public.pem.key" ^
  --private-key-outfile "claim-certs/claim.private.pem.key" ^
  --set-as-active
```

PowerShell

```
aws iot create-keys-and-certificate `
  --certificate-pem-outfile "claim-certs/claim.pem.crt" `
  --public-key-outfile "claim-certs/claim.public.pem.key" `
  --private-key-outfile "claim-certs/claim.private.pem.key" `
  --set-as-active
```

リクエストが成功した場合、レスポンスには証明書に関する情報が含まれます。証明書の ARN は、後で使用できるように書き留めておきます。

3. デバイスが証明書を使用できるようにする AWS IoT ポリシーを作成してアタッチし、固有のデバイス証明書を作成し、フリートプロビジョニングテンプレートを使用してプロビジョニングします。次のポリシーは、デバイスプロビジョニング MQTT API へのアクセスを許可します。詳細については、「AWS IoT Core デベロッパーガイド」の「デバイスプロビジョニング MQTT API」を参照してください。

   以下の操作を実行します。

   a. Greengrass コアデバイスが必要な AWS IoT ポリシードキュメントを含むファイルを作成します。

      例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

      ```
      nano greengrass-provisioning-claim-iot-policy.json
      ```

      次の JSON をファイルにコピーします。

      - *region* の各インスタンスを、フリートプロビジョニングを設定した AWS リージョン に置き換えます。
      - *account-id* の各インスタンスを、AWS アカウント ID に置き換えます。
      - の各インスタンスを、前のセクションで作成したフリートプロビジョニングテンプレートの名前*GreengrassFleetProvisioningTemplate*に置き換えます。

      ```
      {
        "Version": "2012-10-17",
      ```

```
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Receive"
      ],
      "Resource": [
        "arn:aws:iot:region:account-id:topic/$aws/certificates/create/*",
        "arn:aws:iot:region:account-id:topic/$aws/provisioning-
templates/GreengrassFleetProvisioningTemplate/provision/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iot:Subscribe",
      "Resource": [
        "arn:aws:iot:region:account-id:topicfilter/$aws/certificates/create/*",
        "arn:aws:iot:region:account-id:topicfilter/$aws/provisioning-
templates/GreengrassFleetProvisioningTemplate/provision/*"
      ]
    }
  ]
}
```

b. ポリシードキュメントから AWS IoT ポリシーを作成します。

- を、作成するポリシーの名前*GreengrassProvisioningClaimPolicy*に置き換えます。

```
aws iot create-policy --policy-name GreengrassProvisioningClaimPolicy --policy-
document file://greengrass-provisioning-claim-iot-policy.json
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "policyName": "GreengrassProvisioningClaimPolicy",
```

```
  "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassProvisioningClaimPolicy",
  "policyDocument": "{
    \"Version\": \"2012-10-17\",
    \"Statement\": [
      {
        \"Effect\": \"Allow\",
        \"Action\": \"iot:Connect\",
        \"Resource\": \"*\"
      },
      {
        \"Effect\": \"Allow\",
        \"Action\": [
          \"iot:Publish\",
          \"iot:Receive\"
        ],
        \"Resource\": [
          \"arn:aws:iot:region:account-id:topic/$aws/certificates/create/*\",
          \"arn:aws:iot:region:account-id:topic/$aws/provisioning-
templates/GreengrassFleetProvisioningTemplate/provision/*\"
        ]
      },
      {
        \"Effect\": \"Allow\",
        \"Action\": \"iot:Subscribe\",
        \"Resource\": [
          \"arn:aws:iot:region:account-id:topicfilter/$aws/certificates/create/
*\",
          \"arn:aws:iot:region:account-id:topicfilter/$aws/provisioning-
templates/GreengrassFleetProvisioningTemplate/provision/*\"
        ]
      }
    ]
  }",
  "policyVersionId": "1"
}
```

4. プロビジョニングクレーム証明書に AWS IoT ポリシーをアタッチします。

   - をアタッチするポリシーの名前*GreengrassProvisioningClaimPolicy*に置き換えます。
   - ターゲット ARN をプロビジョニングクレーム証明書の ARN に置き換えます。

```
aws iot attach-policy --policy-name GreengrassProvisioningClaimPolicy --
target arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
```

要求が正常に処理された場合、コマンドは出力されません。

これで、デバイスが AWS IoT に登録し、自身を Greengrass のコアデバイスとしてプロビジョニングするために使用できるプロビジョニングクレーム証明書とプライベートキーができました。クレーム証明書とプライベートキーは製造時にデバイスに埋め込むか、AWS IoT Greengrass Core ソフトウェアをインストールする前に証明書とキーをデバイスにコピーすることができます。詳細については、「AWS IoT フリートプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。

AWS IoT フリートプロビジョニングプラグインを設定する

AWS IoT フリートプロビジョニングプラグインでは、次の設定パラメータが提供されており、フリートプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールするときにカスタマイズすることができます。

`rootPath`

AWS IoT Greengrass Core ソフトウェアのルートとして使用するフォルダへのパス。

`awsRegion`

フリートプロビジョニングプラグインが AWS リソースをプロビジョニングするために使用する AWS リージョン。

`iotDataEndpoint`

AWS アカウント の AWS IoT データエンドポイント。

`iotCredentialEndpoint`

AWS IoT の AWS アカウント 認証情報エンドポイント。

`iotRoleAlias`

トークン交換 IAM ロールを指す AWS IoT ロールエイリアス。AWS IoT 認証情報プロバイダは、Greengrass コアデバイスに AWS サービスとやり取りを許可するため、このロースを継承し

ます。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

`provisioningTemplate`

AWS リソースをプロビジョニングするために使用する AWS IoT フリートプロビジョニングテンプレート。このテンプレートでは、以下を指定する必要があります。

- AWS IoT モノのリソース。既存のモノのグループのリストを指定して、オンラインになったときに各デバイスにコンポーネントをデプロイできます。
- AWS IoT ポリシーリソース。このリソースは、次のいずれかのプロパティを定義できます。
  - 既存の AWS IoT ポリシーの名前。このオプションを選択した場合、このテンプレートから作成されたコアデバイスは同じ AWS IoT ポリシーを使用するようになり、そのアクセス許可はフリートとして管理することができます。
  - AWS IoT ポリシードキュメント。このオプションを選択した場合、このテンプレートから作成する各コアデバイスは、一意の AWS IoT ポリシーを使用するようになり、各コアデバイスのアクセス許可を管理することができます。
- AWS IoT 証明書リソース。この証明書リソースでは、`AWS::IoT::Certificate::Id` パラメータを使用して、証明書をコアデバイスにアタッチする必要があります。詳細については、「 AWS IoTデベロッパーガイド」の「J ust-in-time プロビジョニング」を参照してください。

詳細については、「AWS IoT Core デベロッパーガイド」の「プロビジョニングテンプレート」を参照してください。

`claimCertificatePath`

`provisioningTemplate` で指定したプロビジョニングテンプレートのプロビジョニングクレーム証明書へのパス。詳細については、AWS IoT Core API リファレンスの「CreateProvisioningClaim」を参照してください。

`claimCertificatePrivateKeyPath`

`provisioningTemplate` で指定したプロビジョニングテンプレートのプロビジョニングクレーム証明書プライベートキーへのパス。詳細については、AWS IoT Core API リファレンスの「CreateProvisioningClaim」を参照してください。

> **Important**
>
> プロビジョニングクレームプライベートキーは、Greengrass コアデバイス上にある場合を含め、常に保護する必要があります。Amazon CloudWatch メトリクスとログを使用して、デバイスをプロビジョニングするためのクレーム証明書の不正使用などの誤用の兆

候をモニタリングすることをお勧めします。悪用を検出した場合は、プロビジョニングクレーム証明書を無効にして、デバイスのプロビジョニングに使用できないようにします。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT のモニタリング」を参照してください。
デバイス数と AWS アカウント に自身を登録するデバイスについてより適切に管理できるように、フリートプロビジョニングテンプレートを作成するときに、事前プロビジョニングフックを指定することができます。事前プロビジョニングフックは、登録時にデバイスが提供するテンプレートパラメータを検証する AWS Lambda 関数です。例えば、デバイス ID をデータベースと照合して、デバイスにプロビジョニングする権限があることを確認する事前プロビジョニングフックを作成することができます。詳細については、「AWS IoT Core デベロッパーガイド」の「事前プロビジョニングフック」を参照してください。

`rootCaPath`

Amazon ルート認証局 (CA) 証明書へのパス。

`templateParameters`

(オプション) フリートプロビジョニングテンプレートに提供するパラメータのマップ。詳細については、「AWS IoT Core デベロッパーガイド」の「プロビジョニングテンプレートのパラメータセクション」を参照してください。

`deviceId`

(オプション) フリートプロビジョニングプラグインが AWS IoT への MQTT 接続を作成するときに、クライアント ID として使用するデバイス識別子。

デフォルト: ランダム値の UUID。

`mqttPort`

(オプション) MQTT 接続に使用するポート。

デフォルト: 8883

`proxyUrl`

(オプション) プロキシサーバーの URL (`scheme://userinfo@host:port` 形式)。HTTPS プロキシを使用するには、フリートプロビジョニングプラグインのバージョン 1.1.0 以降を使用する必要があります。

- `scheme` - スキーム。`http` または `https` である必要があります。

  > **Important**
  >
  > HTTPS プロキシを使用するには、Greengrass コアデバイスで Greengrass nucleus v2.5.0 以降を実行している必要があります。
  > HTTPS プロキシを設定する場合は、コアデバイスの Amazon ルート CA 証明書にプロキシサーバー CA 証明書を追加する必要があります。詳細については、「コアデバイスが HTTPS プロキシを信頼できるようにする」を参照してください。

- `userinfo` - (オプション) ユーザー名とパスワードの情報。この情報を `url` で指定する場合、Greengrass コアデバイスは `username` および `password` フィールドを無視します。
- `host` – プロキシサーバーのホスト名または IP アドレス。
- `port` - (オプション) ポート番号。ポートを指定しない場合、Greengrass コアデバイスは次のデフォルト値を使用します。
  - `http` – 80
  - `https` – 443

`proxyUserName`

(オプション) プロキシサーバーを認証するユーザー名です。

`proxyPassword`

(オプション) プロキシサーバーを認証するユーザー名です。

csrPath

(オプション) CSR からデバイス証明書を作成するために使用する、証明書署名要求 (CSR) ファイルへのパスです。詳細については、「AWS IoT Core デベロッパーガイド」の「クレームによるプロビジョニング」を参照してください。

csrPrivateKeyパス

(オプション、`csrPath` 宣言されている場合は必須) CSR の生成に使用されるプライベートキーへのパスです。CSR の生成は、プライベートキーを使用して行われている必要があります。詳細については、「AWS IoT Core デベロッパーガイド」の「クレームによるプロビジョニング」を参照してください。

AWS IoT フリートプロビジョニングプラグインの変更ログ

次の表は、クレームプラグイン (`aws.greengrass.FleetProvisioningByClaim`) による AWS IoT フリートプロビジョニングの各バージョンの変更を説明したものです。

バージョン	変更
1.2.0	バグ修正と機能向上 • プライベートキーの設定可能なパスを使用した証明書署名リクエストを介するデバイスプロビジョニングのサポートを追加しました。 • 軽微な修正と改善。
1.1.0	バグ修正と機能向上 • Windows デバイスでプラグインを設定するときの、追加のファイルパス形式へのサポートが追加されました。 • HTTPS ネットワークプロキシ設定へのサポートが追加されました。詳細については、ポート 443 での接続またはネットワークプロキシを通じた接続 および コアデバイスが HTTPS プロキシを信頼できるようにする を参照してください。
1.0.0	当初のバージョン

カスタムリソースプロビジョニングを使用して AWS IoT Greengrass Core ソフトウェアをインストールする

この機能は、Greengrass nucleus コンポーネントの v2.4.0 以降に利用できます。

AWS IoT Greengrass Core ソフトウェアインストーラは、必要な AWS リソースをプロビジョニングするカスタムプラグインに実装できる Java インターフェイスを提供します。プロビジョニングプラグインを開発して、カスタム X.509 クライアント証明書を使用するか、他のインストールプロセスでサポートされていない複雑なプロビジョニング手順を実行することができます。詳細については、「AWS IoT Core デベロッパーガイド」の「独自のクライアント証明書を作成する」を参照してください。

AWS IoT Greengrass Core ソフトウェアのインストール時にカスタムプロビジョニングプラグインを実行するには、インストーラに提供する JAR ファイルを作成します。インストーラはプラグイン

を実行し、プラグインは Greengrass コアデバイスの AWS リソースを定義するプロビジョニング設定を返します。インストーラはこの情報を使用して、デバイス上の AWS IoT Greengrass Core ソフトウェアを設定します。詳細については、「カスタムプロビジョニングプラグインを開発する」を参照してください。

> **Important**
>
> AWS IoT Greengrass Core ソフトウェアをダウンロードする前に、コアデバイスが AWS IoT Greengrass Core ソフトウェア v2.0 をインストールして実行するための要件を満たしていることを確認してください。

トピック

前提条件

カスタムプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールするには、以下が必要です。

- `DeviceIdentityInterface` を実装するカスタムプロビジョニングプラグインの JAR ファイル。カスタムプロビジョニングプラグインは、各システムに対する値と nucleus 設定パラメータを返す必要があります。これらの値が返されない場合には、インストール時に設定ファイルでこれらの値を提供する必要があります。詳細については、「カスタムプロビジョニングプラグインを開発する」を参照してください。

デバイス環境をセットアップする

このセクションのステップに従って、 AWS IoT Greengrass コアデバイスとして使用する Linux または Windows デバイスをセットアップします。

Linux デバイスをセットアップする

の Linux デバイスをセットアップするには AWS IoT Greengrass V2

1. AWS IoT Greengrass Core ソフトウェアが実行する必要がある Java ランタイムをインストールします。Amazon Corretto または OpenJDK の長期サポートバージョンを使用することをお勧めします。バージョン 8 以降が必要です。次のコマンドは、デバイスに OpenJDK をインストールする方法を示しています。

   - Debian ベースまたは Ubuntu ベースのディストリビューションの場合:

     ```
     sudo apt install default-jdk
     ```

   - Red Hat ベースのディストリビューションの場合 :

     ```
     sudo yum install java-11-openjdk-devel
     ```

   - 複数 Amazon Linux 2:

     ```
     sudo amazon-linux-extras install java-openjdk11
     ```

   - 複数 Amazon Linux 2023:

     ```
     sudo dnf install java-11-amazon-corretto -y
     ```

   インストールが完了したら、次のコマンドを実行して Java が Linux デバイスで実行されていることを確認します。

   ```
   java -version
   ```

   このコマンドは、デバイス上で実行されている Java のバージョンを出力します。例えば、Debian ベースのディストリビューションでは、出力は次のサンプルのようになります。

   ```
   openjdk version "11.0.9.1" 2020-11-04
   OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
   OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
   ```

2. (オプション) デバイスにコンポーネントを実行するデフォルトのシステムユーザーおよびグループを作成します。インストーラ引`--component-default-user`数を使用して、インストール

中に AWS IoT Greengrass Core ソフトウェアインストーラにこのユーザーとグループを作成させることもできます。詳細については、「インストーラ引数」を参照してください。

```
sudo useradd --system --create-home ggc_user
sudo groupadd --system ggc_group
```

3. AWS IoT Greengrass Core ソフトウェアを実行するユーザー (通常は `root`) に、任意のユーザーおよび任意のグループsudoで実行するアクセス許可があることを確認します。

   a. `/etc/sudoers` ファイルを開くには、次のコマンドを実行します。

   ```
   sudo visudo
   ```

   b. ユーザーの権限が次の例のようになっていることを確認します。

   ```
   root    ALL=(ALL:ALL) ALL
   ```

4. (オプション) コンテナ化された Lambda 関数を実行するには、cgroups v1 を有効にし、メモリとデバイスの cgroups を有効にしてマウントする必要があります。コンテナ化された Lambda 関数を実行する予定がない場合、この手順を省略できます。

   これらの cgroups オプションを有効にするには、次の Linux カーネルパラメータを使用してデバイスを起動します。

   ```
   cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
   ```

   デバイスのカーネルパラメータを確認および設定するための情報については、オペレーティングシステムおよびブートローダーのドキュメントを参照してください。指示に従って、カーネルパラメータを永続的に設定します。

5. デバイスの要件 にある要件リストで示されているように、その他の必要となる依存関係をすべてデバイスにインストールします。

Windows デバイスをセットアップする

> **Note**
>
> この機能は、Greengrass nucleus コンポーネントの v2.5.0 以降に利用できます。

の Windows デバイスをセットアップするには AWS IoT Greengrass V2

1. AWS IoT Greengrass Core ソフトウェアが実行する必要がある Java ランタイムをインストールします。Amazon Corretto または OpenJDK の長期サポートバージョンを使用することをお勧めします。バージョン 8 以降が必要です。
2. PATH システム変数で Java が使用可能か確認し、そうでない場合は追加します。 LocalSystem アカウントは AWS IoT Greengrass Core ソフトウェアを実行するため、ユーザーの PATH ユーザー変数の代わりに Java を PATH システム変数に追加する必要があります。以下の操作を実行します。

   a. Windows キーを押してスタートメニューを開きます。
   b. **`environment variables`** を入力して、スタートメニューからシステムオプションを検索します。
   c. スタートメニューの検索結果から [Edit the system environment variables] (システム環境変数を編集) をクリックして、[System properties] (システムプロパティ) ウィンドウを開きます。
   d. [Environment variables...] (環境変数...) を選択して、[Environment Variables] (環境可変) ウィンドウを開きます。
   e. [System variables] (システム変数) で、[Path] (パス) 、[Edit] (編集) の順に選択します。[Edit environment variable] (環境変数の編集) ウィンドウでは、個別の行に各パスを表示できます。
   f. Java インストールの `bin` フォルダへのパスが存在しているかを確認します。このパスは、次の例のように表示されます。

      ```
      C:\\Program Files\\Amazon Corretto\\jdk11.0.13_8\\bin
      ```

   g. [Path] (パス) で Java インストールの `bin` フォルダが で見つからない場合は、[New] (新規) を選択してこれを追加した上で、[OK] を選択します。
3. 管理者として Windows コマンドプロンプト `cmd.exe` を開きます。
4. Windows デバイスの LocalSystem アカウントにデフォルトユーザーを作成します。*#####*を安全なパスワードに置き換えます。

   ```
   net user /add ggc_user password
   ```

> **Tip**
>
> Windows の構成によっては、ユーザーのパスワードの期限切れが、将来の日付に設定されている場合があります。Greengrass アプリケーションの動作を継続させるためには、パスワードの有効期限を追跡し、その期限が切れる前に更新します。ユーザーのパスワードには、期限切れを起こさないような設定も可能です。
>
> - ユーザーとパスワードの有効期限を確認するには、次のコマンドを実行します。
>
>   ```
>   net user ggc_user | findstr /C:expires
>   ```
>
> - ユーザーのパスワードが期限切れにならないように設定するには、次のコマンドを実行します。
>
>   ```
>   wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
>   ```
>
> - wmic コマンドが廃止された Windows 10 以降を使用している場合は、次の PowerShell コマンドを実行します。
>
>   ```
>   Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
>    'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
>   ```

5. デバイスに Microsoft から PsExec ユーティリティをダウンロードしてインストールします。
6. PsExec ユーティリティを使用して、デフォルトユーザーのユーザー名とパスワードを LocalSystem アカウントの認証情報マネージャーインスタンスに保存します。*#####*を以前に設定したユーザーのパスワードに置き換えます。

   ```
   psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
   ```

   PsExec License Agreement が開いたら、Accept を選択し、ライセンスに同意してコマンドを実行します。

> **Note**
>
> Windows デバイスでは、 LocalSystem アカウントが Greengrass nucleus を実行するため、 PsExec ユーティリティを使用してデフォルトのユーザー情報を LocalSystem アカウントに格納する必要があります。認証情報マネージャーアプリケーションを使用

すると、この情報は アカウントではなく、現在ログオンしているユーザーの Windows LocalSystem アカウントに保存されます。

AWS IoT Greengrass Core ソフトウェアをダウンロードする

AWS IoT Greengrass Core ソフトウェアの最新バージョンは、次の場所からダウンロードできます。

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアの特定のバージョンは、次の場所からダウンロードできます。*#####*をダウンロードするバージョンに置き換えます。
>
> ```
> https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip
> ```

AWS IoT Greengrass Core ソフトウェアをダウンロードするには

1. コアデバイスで、 AWS IoT Greengrass Core ソフトウェアを という名前のファイルにダウンロードしますgreengrass-nucleus-latest.zip。

   Linux or Unix

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip > greengrass-nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip > greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip -OutFile greengrass-nucleus-latest.zip
   ```

このソフトウェアをダウンロードすると、Greengrass Core ソフトウェアのライセンス契約に同意したものと見なされます。

2. (オプション) Greengrass nucleus ソフトウェア署名を確認するには

> **Note**
>
> この機能は、Greengrass nucleus バージョン 2.9.5 以降で使用できます。

a. 以下のコマンドを使用して、Greengrass nucleus アーティファクトの署名を確認します。

Linux or Unix

```
jarsigner -verify -certs -verbose greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

インストールする JDK のバージョンによって、ファイル名が異なる場合があります。インストールした JDK のバージョンに *jdk17.0.6_10* を置き換えてください。

```
"C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe" -
verify -certs -verbose greengrass-nucleus-latest.zip
```

PowerShell

インストールする JDK のバージョンによって、ファイル名が異なる場合があります。インストールした JDK のバージョンに *jdk17.0.6_10* を置き換えてください。

```
'C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe' -
verify -certs -verbose greengrass-nucleus-latest.zip
```

b. `jarsigner` が起動すると、検証結果を示す出力が得られます。

i. Greengrass nucleus の zip ファイルに署名されると、出力に以下のような文が表示されます：

```
jar verified.
```

ii. Greengrass nucleus の zip ファイルに署名されないと、出力に以下のような文が表示されます：

```
jar is unsigned.
```

c. Jarsigner `-certs` を `-verify` と `-verbose` オプションと一緒に提供した場合、出力には署名者証明書の詳細情報も含まれます。

3. AWS IoT Greengrass Core ソフトウェアをデバイス上のフォルダに解凍します。を使用するフォルダ*GreengrassInstaller*に置き換えます。

Linux or Unix

```
unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-
nucleus-latest.zip
```

Windows Command Prompt (CMD)

```
mkdir GreengrassInstaller && tar -xf greengrass-nucleus-latest.zip -
C GreengrassInstaller && del greengrass-nucleus-latest.zip
```

PowerShell

```
Expand-Archive -Path greengrass-nucleus-latest.zip -DestinationPath .\
\GreengrassInstaller
rm greengrass-nucleus-latest.zip
```

4. （オプション) 次のコマンドを実行して、 AWS IoT Greengrass Core ソフトウェアのバージョンを確認します。

```
java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
```

> **Important**
>
> v2.4.0 より前のバージョンの Greengrass nucleus をインストールする場合は、 AWS IoT Greengrass Core ソフトウェアをインストールした後にこのフォルダを削除しないでください。 AWS IoT Greengrass Core ソフトウェアは、このフォルダ内のファイルを使用して実行します。

ソフトウェアの最新バージョンをダウンロードした場合は、v2.4.0 以降をインストールし、AWS IoT Greengrass Core ソフトウェアをインストールした後にこのフォルダを削除できます。

AWS IoT Greengrass Core ソフトウェアのインストール

次のアクションを指定する引数を含むインストーラを実行します。

- カスタムプロビジョニングプラグインを使用して AWS リソースをプロビジョニングするように指定する部分設定ファイルからインストールします。 AWS IoT Greengrass Core ソフトウェアは、デバイス上のすべての Greengrass コンポーネントの設定を指定する設定ファイルを使用します。インストーラは、指定した部分設定ファイルと、カスタムプロビジョニングプラグインが作成する AWS リソースから完全な設定ファイルを作成します。
- コアデバイスでソフトウェアコンポーネントを実行するために `ggc_user` システムユーザーを使用するように指定します。Linux デバイスでは、このコマンドも `ggc_group` システムグループを使用するように指定し、さらにインストーラによってシステムユーザーとグループが、ユーザーに代わって作成されます。
- AWS IoT Greengrass Core ソフトウェアを、起動時に実行されるシステムサービスとしてセットアップします。Linux デバイスでは、これは Systemd init システムが必要です。

> **Important**
>
> Windows コアデバイスでは、 AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります。

指定できる引数の詳細については、「インストーラ引数」を参照してください。

> **Note**
>
> メモリが制限された AWS IoT Greengrass デバイスで実行している場合は、 AWS IoT Greengrass Core ソフトウェアが使用するメモリ量を制御できます。メモリ割り当てを制御するには、nucleus コンポーネントの `jvmOptions` 設定パラメータで JVM ヒープのサイズオプションを設定できます。詳細については、「JVM オプションでメモリ割り当てを制御する」を参照してください。

AWS IoT Greengrass Core ソフトウェアをインストールするには (Linux)

1. AWS IoT Greengrass Core ソフトウェアのバージョンを確認します。

   - を、ソフトウェアを含むフォルダへのパス*GreengrassInstaller*に置き換えます。

   ```
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

2. テキストエディタを使用し、config.yaml という名前の設定ファイルを作成してインストーラに提供します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

   ```
   nano GreengrassInstaller/config.yaml
   ```

   次の YAML コンテンツをファイルにコピーします。

   ```
   ---
   system:
     rootpath: "/greengrass/v2"
     # The following values are optional. Return them from the provisioning plugin or
    set them here.
     # certificateFilePath: ""
     # privateKeyPath: ""
     # rootCaPath: ""
     # thingName: ""
   services:
     aws.greengrass.Nucleus:
       version: "2.12.2"
       configuration:
         # The following values are optional. Return them from the provisioning plugin
    or set them here.
         # awsRegion: ""
         # iotRoleAlias: ""
         # iotDataEndpoint: ""
         # iotCredEndpoint: ""
     com.example.CustomProvisioning:
       configuration:
         # You can specify configuration parameters to provide to your plugin.
         # pluginParameter: ""
   ```

次に、以下の操作を実行します。

- *2.12.2* を AWS IoT Greengrass Core ソフトウェアのバージョンに置き換えます。
- の各インスタンスを Greengrass ルートフォルダ*/greengrass/v2*に置き換えます。
- (オプション) システムおよび nucleus の設定値を指定します。プロビジョニングプラグインから提供されない場合には、これらの値を設定する必要があります。
- (オプション) プロビジョニングプラグインに提供する設定パラメータを指定します。

> **Note**
>
> この設定ファイルでは、次の例で示されているように、使用するポートやネットワークプロキシなどの他の設定オプションをカスタマイズすることができます。詳細については、Greengrass nucleus 設定を参照してください。

```
---
system:
  rootpath: "/greengrass/v2"
  # The following values are optional. Return them from the provisioning
 plugin or set them here.
  # certificateFilePath: ""
  # privateKeyPath: ""
  # rootCaPath: ""
  # thingName: ""
services:
  aws.greengrass.Nucleus:
    version: "2.12.2"
    configuration:
      mqtt:
        port: 443
      greengrassDataPlanePort: 443
      networkProxy:
        noProxyAddresses: "http://192.168.0.1,www.example.com"
        proxy:
          url: "http://my-proxy-server:1100"
          username: "Mary_Major"
          password: "pass@word1357"
      # The following values are optional. Return them from the provisioning
 plugin or set them here.
      # awsRegion: ""
      # iotRoleAlias: ""
```

```
      # iotDataEndpoint: ""
      # iotCredEndpoint: ""
  com.example.CustomProvisioning:
    configuration:
      # You can specify configuration parameters to provide to your plugin.
      # pluginParameter: ""
```

3. インストーラーを実行します。--trusted-plugin を指定してカスタムプロビジョニングプラグインを提供し、--init-config を指定して設定ファイルを提供します。

   - */greengrass/v2* または *C:\greengrass\v2* を Greengrass ルートフォルダに置き換えます。
   - の各インスタンスを、インストーラを解凍したフォルダ*GreengrassInstaller*に置き換えます。
   - カスタムプロビジョニングプラグイン JAR ファイルへのパスを、プラグインの JAR ファイルへのパスに置き換えます。

   Linux or Unix

   ```
   sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
     -jar ./GreengrassInstaller/lib/Greengrass.jar \
     --trusted-plugin /path/to/com.example.CustomProvisioning.jar \
     --init-config ./GreengrassInstaller/config.yaml \
     --component-default-user ggc_user:ggc_group \
     --setup-system-service true
   ```

   Windows Command Prompt (CMD)

   ```
   java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" ^
     -jar ./GreengrassInstaller/lib/Greengrass.jar ^
     --trusted-plugin /path/to/com.example.CustomProvisioning.jar ^
     --init-config ./GreengrassInstaller/config.yaml ^
     --component-default-user ggc_user ^
     --setup-system-service true
   ```

   PowerShell

   ```
   java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" `
     -jar ./GreengrassInstaller/lib/Greengrass.jar `
   ```

```
  --trusted-plugin /path/to/com.example.CustomProvisioning.jar `
  --init-config ./GreengrassInstaller/config.yaml `
  --component-default-user ggc_user `
  --setup-system-service true
```

> **Important**
>
> Windows コアデバイスでは、 を指定`--setup-system-service true`して AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります。

`--setup-system-service true` を指定する場合、ソフトウェアをシステムサービスとしてセットアップして実行したら、インストーラは `Successfully set up Nucleus as a system service` を出力します。それ以外の場合、正常にソフトウェアがインストールされていれば、インストーラはメッセージを出力しません。

> **Note**
>
> `--provision true` 引数なしでインストーラを実行するとき、ローカル開発ツールをデプロイするために `deploy-dev-tools` 引数を使用できません。Greengrass CLI をデバイスに直接デプロイする方法の情報については、「Greengrass コマンドラインインターフェイス」を参照してください。

4. ルートフォルダのファイルを確認して、インストールを確認します。

   Linux or Unix

   ```
   ls /greengrass/v2
   ```

   Windows Command Prompt (CMD)

   ```
   dir C:\greengrass\v2
   ```

   PowerShell

   ```
   ls C:\greengrass\v2
   ```

インストールが正常に処理された場合、ルートフォルダには config、packages、logs などの複数のフォルダが含まれます。

AWS IoT Greengrass Core ソフトウェアをシステムサービスとしてインストールした場合、インストーラがソフトウェアを実行します。それ以外の場合、ソフトウェアを手動で実行する必要があります。詳細については、「AWS IoT Greengrass Core ソフトウェアを実行する」を参照してください。

ソフトウェアと を設定して使用する方法の詳細については AWS IoT Greengrass、以下を参照してください。

- AWS IoT Greengrass Core ソフトウェアを設定する
- AWS IoT Greengrass コンポーネントを開発する
- デバイスに AWS IoT Greengrass コンポーネントのデプロイ
- Greengrass コマンドラインインターフェイス

カスタムプロビジョニングプラグインを開発する

カスタムプロビジョニングプラグインを開発するには、com.aws.greengrass.provisioning.DeviceIdentityInterface インターフェイスを実装する Java クラスを作成します。Greengrass nucleus JAR ファイルをプロジェクトに含めて、このインターフェイスとクラスにアクセスすることができます。このインターフェイスは、プラグイン設定を入力し、プロビジョニング設定を出力するメソッドを定義するためのものです。プロビジョニング設定では、システムと Greengrass nucleus コンポーネントの設定を定義します。AWS IoT Greengrass コアソフトウェアインストーラは、このプロビジョニング設定を使用して、デバイス上の AWS IoT Greengrass Core コアソフトウェアを設定します。

カスタムプロビジョニングプラグインを開発したら、JAR ファイルとして構築し、インストール中にプラグインを実行するために、AWS IoT Greengrass コアソフトウェアインストーラに提供します。インストーラは、インストーラが使用するものと同じ JVM でカスタムプロビジョニングプラグインを実行するため、プラグインコードのみが含まれた JAR を作成することができます。

> **Note**
>
> AWS IoT フリートプロビジョニングプラグインは、インストール中にフリートプロビジョニングを使用するため、DeviceIdentityInterface を実装します。フリートプロビジョニ

ングプラグインはオープンソースであるため、ソースコードを確認して、プロビジョニングプラグインインターフェイスの使用方法例を見ることができます。詳細については、GitHub の「AWS IoT フリートプロビジョニングプラグイン」を参照してください。

トピック

要件

カスタムプロビジョニングプラグインを開発するには、次の要件を満たす Java クラスを作成する必要があります。

- `com.aws.greengrass` パッケージ、または `com.aws.greengrass` パッケージ内のパッケージを使用します。
- 引数を持たないコンストラクタが必要です。
- `DeviceIdentityInterface` インターフェイスを実装します。詳細については、「DeviceIdentityInterface インターフェイスを実装する」を参照してください。

DeviceIdentityInterface インターフェイスを実装する

カスタムプラグインで `com.aws.greengrass.provisioning.DeviceIdentityInterface` インターフェースを使用するには、Greengrass nucleus をプロジェクトの依存関係として追加します。

カスタムプロビジョニングプラグインプロジェクトで DeviceIdentityInterface を使用するには

- Greengrass nucleus JAR ファイルをライブラリとして追加するか、Greengrass nucleus を Maven の依存関係として追加できます。次のいずれかを実行します。
  - Greengrass nucleus JAR ファイルをライブラリとして追加するには、Greengrass nucleus JAR を含む AWS IoT Greengrass コアソフトウェアをダウンロードします。AWS IoT Greengrass コアソフトウェアの最新バージョンを次の場所からダウンロードできます。
    - https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

Greengrass nucleus JAR ファイル (`Greengrass.jar`) は、ZIP ファイル内の `lib` フォルダにあります。この JAR ファイルをプロジェクトに追加します。

- Maven プロジェクトで Greengrass nucleus を使用するには、`com.aws.greengrass` グループ内の `nucleus` アーティファクトに依存関係を追加します。また、Greengrass nucleus は Maven Central リポジトリでは利用できないため、`greengrass-common` レポジトリも追加する必要があります。

```
<project ...>
    ...
    <repositories>
        <repository>
            <id>greengrass-common</id>
            <name>greengrass common</name>
            <url>https://d2jrmugq4soldf.cloudfront.net/snapshots</url>
        </repository>
    </repositories>
    ...
    <dependencies>
        <dependency>
            <groupId>com.aws.greengrass</groupId>
            <artifactId>nucleus</artifactId>
            <version>2.5.0-SNAPSHOT</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

DeviceIdentityInterface インターフェイス

`com.aws.greengrass.provisioning.DeviceIdentityInterface` インターフェイスには、次のシェイプがあります。

> **Note**
>
> これらのクラスは、GitHub にある Greengrass nucleus source code の com.aws.greengrass.provisioning package でも確認できます。

```
public interface com.aws.greengrass.provisioning.DeviceIdentityInterface {
    ProvisionConfiguration updateIdentityConfiguration(ProvisionContext context)
            throws RetryableProvisioningException, InterruptedException;

    // Return the name of the plugin.
    String name();
}

com.aws.greengrass.provisioning.ProvisionConfiguration {
    SystemConfiguration systemConfiguration;
    NucleusConfiguration nucleusConfiguration
}

com.aws.greengrass.provisioning.ProvisionConfiguration.SystemConfiguration {
    String certificateFilePath;
    String privateKeyPath;
    String rootCAPath;
    String thingName;
}

com.aws.greengrass.provisioning.ProvisionConfiguration.NucleusConfiguration {
    String awsRegion;
    String iotCredentialsEndpoint;
    String iotDataEndpoint;
    String iotRoleAlias;
}

com.aws.greengrass.provisioning.ProvisioningContext {
    Map<String, Object> parameterMap;
    String provisioningPolicy;  // The policy is always "PROVISION_IF_NOT_PROVISIONED".
}

com.aws.greengrass.provisioning.exceptions.RetryableProvisioningException {}
```

SystemConfiguration と NucleusConfiguration の各設定値は、AWS IoT Greengrass コアソフトウェアのインストールに必要ですが、null を戻すことができます。カスタムプロビジョニングプラグインがいずれかの設定値に null を返した場合、AWS IoT Greengrass コアソフトウェアのインストーラに提供する config.yaml ファイルを作成する際に、システムまたは Nucleus の設定にその値を提供する必要があります。config.yaml で定義したオプションに対して、カスタムプロビジョニングプラグインが NULL 以外の値を返した場合、インストーラは config.yaml の値をプラグインから返された値に置き換えます。

インストーラ引数

AWS IoT Greengrass Core ソフトウェアには、ソフトウェアをセットアップし、Greengrass コアデバイスを実行するのに必要となる AWS リソースをプロビジョニングするインストーラが含まれています。インストーラには、インストールを設定するために指定できる次の引数が含まれます。

`-h, --help`

(オプション) インストーラのヘルプ情報を表示します。

`--version`

(オプション) AWS IoT Greengrass Core ソフトウェアのバージョンを表示します。

`-Droot`

(オプション) AWS IoT Greengrass Core ソフトウェアのルートとして使用するフォルダへのパス。

> **Note**
>
> この引数は JVM プロパティを設定するため、インストーラを実行するときに `-jar` より前に指定する必要があります。例えば、`java -Droot="/greengrass/v2" -jar /path/to/Greengrass.jar` と指定します。

デフォルト:

- Linux: `~/.greengrass`
- Windows: `%USERPROFILE%/.greengrass`

`-ar, --aws-region`

AWS IoT Greengrass Core ソフトウェアが、必要な AWS リソースを取得または作成するために使用する AWS リージョン。

`-p, --provision`

(オプション) このデバイスをAWS IoT モノとして登録し、コアデバイスが必要とする AWS リソースをプロビジョニングすることができます。`true` を指定すると、AWS IoT Greengrass Core ソフトウェアは、AWS IoT モノ、(オプションの) AWS IoT モノグループ、IAM ロール、および AWS IoT ロールエイリアスをプロビジョニングします。

デフォルト: `false`

`-tn, --thing-name`

(オプション) このコアデバイスとして登録する AWS IoT モノの名前。モノの名前が AWS アカウント に存在しない場合、AWS IoT Greengrass Core ソフトウェアが作成します。

> **Note**
>
> モノの名前にコロン (:) 記号を含むことができません。

この引数を適用するには、`--provision true` を指定する必要があります。

デフォルト: `GreengrassV2IotThing_` とランダムな UUID。

`-tgn, --thing-group-name`

(オプション) このコアデバイスの AWS IoT モノを追加する AWS IoT モノグループの名前。デプロイがこのモノグループをターゲットにしている場合、このコアデバイスは、AWS IoT Greengrass に接続したときにそのデプロイを受け取ります。この名前のモノグループが AWS アカウント に存在しない場合、AWS IoT Greengrass Core ソフトウェアが作成します。

> **Note**
>
> モノグループ名にコロン (:) 記号を含めることはできません。

この引数を適用するには、`--provision true` を指定する必要があります。

`-tpn, --thing-policy-name`

この機能は、Greengrass nucleus コンポーネントの v2.4.0 以降に利用できます。

(オプション) このコアデバイスの AWS IoT モノ証明書にアタッチする AWS IoT ポリシーの名前。この名前の AWS IoT ポリシーが AWS アカウント に存在しない場合、AWS IoT Greengrass Core ソフトウェアが作成します。

AWS IoT Greengrass Core ソフトウェアでは、デフォルトで寛容な AWS IoT ポリシーが作成されます。このポリシーの範囲を絞り込むか、ユースケースに応じた権限に制限するカスタムポリシーを作成することができます。詳細については、「AWS IoT Greengrass V2 コアデバイス向けの最低限の AWS IoT ポリシー」を参照してください。

この引数を適用するには、`--provision true` を指定する必要があります。

デフォルト: `GreengrassV2IoTThingPolicy`

`-trn`, `--tes-role-name`

(オプション) コアデバイスが AWS サービスとやり取りするための AWS 認証情報を取得するために使用する IAM ロールの名前。この名前のロールが AWS アカウント に存在しない場合、AWS IoT Greengrass Core ソフトウェアが `GreengrassV2TokenExchangeRoleAccess` ポリシーを使ってロールを作成します。このロールは、コンポーネントのアーティファクトをホストする S3 バケットにはアクセスできません。そのため、コンポーネントを作成するときに、アーティファクトの S3 バケットとオブジェクトへのアクセス許可を追加する必要があります。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

この引数を適用するには、`--provision true` を指定する必要があります。

デフォルト: `GreengrassV2TokenExchangeRole`

`-tra`, `--tes-role-alias-name`

(オプション) このコアデバイスの AWS 認証情報を提供する IAM ロールを指している AWS IoT ロールエイリアスの名前。この名前のロールエイリアスが AWS アカウント に存在しない場合、AWS IoT Greengrass Core ソフトウェアはそれを作成し、指定されている IAM ロールを指すように設定します。

この引数を適用するには、`--provision true` を指定する必要があります。

デフォルト: `GreengrassV2TokenExchangeRoleAlias`

`-ss`, `--setup-system-service`

(オプション) AWS IoT Greengrass Core ソフトウェアは、このデバイスの起動時に実行されるシステムサービスとして設定することができます。システムサービス名は `greengrass` です。詳細については、「Greengrass nucleus をシステムサービスとして設定する」を参照してください。

Linux オペレーティングシステムでこの引数を使用する場合、systemd init システムがデバイス上で利用できる必要があります。

> **Important**
>
> Windows コアデバイスでは、AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります

デフォルト: `false`

`-u, --component-default-user`

AWS IoT Greengrass Core ソフトウェアがコンポーネントを実行するために使用するユーザーの名前または ID。例えば、**`ggc_user`** と指定することができます。この値は、Windows オペレーティングシステムでインストーラを実行するときに必要となります。

Linux オペレーティングシステムでは、オプションでグループを指定することもできます。ユーザーとグループをコロンで区切って指定します。例: **`ggc_user:ggc_group`**。

Linux オペレーティングシステムの場合、次の点についても考慮する必要があります。

- root として実行する場合、デフォルトのコンポーネントユーザーが設定ファイルで定義されたユーザーになります。設定ファイルでユーザーが定義されていない場合、デフォルトの `ggc_user:ggc_group` になります。もし `ggc_user` または `ggc_group` が存在しない場合は、ソフトウェアが作成します。
- 非ルートユーザーとして実行すると、AWS IoT Greengrass Core ソフトウェアはそのユーザーとしてコンポーネントを実行します。
- グループを指定しなかった場合、AWS IoT Greengrass Core ソフトウェアは、システムユーザーのプライマリグループを使用します。

詳細については、「コンポーネントを実行するユーザーを設定する」を参照してください。

`-d, --deploy-dev-tools`

(オプション) Greengrass CLI コンポーネントをダウンロードして、このコアデバイスにデプロイできます。このツールを使用して、このコアデバイス上でコンポーネントを開発したりデバッグしたりできます。

> **Important**
>
> このコンポーネントは、本番環境ではなく、開発環境でのみで使用することをお勧めします。このコンポーネントは、通常、本番環境では必要とされない情報や操作へのアクセスを提供します。このコンポーネントを必要なコアデバイスにのみデプロイして、最小特権の原則に従います。

この引数を適用するには、`--provision true` を指定する必要があります。

デフォルト: `false`

`-init, --init-config`

(オプション) AWS IoT Greengrass Core ソフトウェアのインストールに使用する設定ファイルへのパス。このオプションを使用すると、特定の nucleus 設定された新しいコアデバイスなどを、設定することができます。

> **Important**
>
> 指定した設定ファイルとコアデバイス上の既存の設定ファイルがマージされます。これには、コアデバイス上のコンポーネントとコンポーネント設定が含まれます。設定ファイルには、変更しようとしている設定だけをリストすることをおすすめします。

`-tp, --trusted-plugin`

(オプション) 信頼されたプラグインとして読み込む JAR ファイルへのパス。このオプションは、フリープロビジョニングまたはカスタムプロビジョニングでインストールする場合、またはハードウェアセキュリティモジュールでプライベートキーと証明書を使用してインストールする場合など、プロビジョニングプラグインの JAR ファイルを提供するために使用します。

`-s, --start`

(オプション) AWS IoT Greengrass Core ソフトウェアのインストール後に起動し、オプションでリソースをプロビジョニングすることができます。

デフォルト: `true`

AWS IoT Greengrass Core ソフトウェアを実行する

AWS IoT Greengrass Core ソフトウェアをインストールしたら、それを実行してデバイスを AWS IoT Greengrass に接続します。

AWS IoT Greengrass Core ソフトウェアをインストールするときに、systemd を使用してシステムサービスとしてインストールするかどうかを指定できます。このオプションを選択すると、インストーラがソフトウェアを実行し、デバイスの起動時に実行するように設定します。

> **Important**
>
> Windows コアデバイスでは、AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります

トピック

AWS IoT Greengrass Core ソフトウェアがシステムサービスとして実行されているかどうかを確認する

AWS IoT Greengrass Core ソフトウェアをインストールするときに、`--setup-system-service true` 引数を指定して、AWS IoT Greengrass Core ソフトウェアをシステムサービスとしてインストールできます。Linux デバイスでは、systemd init システムで AWS IoT Greengrass Core ソフトウェアをシステムサービスとしてセットアップする必要があります。このオプションを使用すると、インストーラがソフトウェアを実行し、デバイスの起動時に実行するように設定します。インストーラは、AWS IoT Greengrass Core ソフトウェアをシステムサービスとして正常にインストールすると、次のメッセージを出力します。

```
Successfully set up Nucleus as a system service
```

以前にAWS IoT Greengrass Core ソフトウェアをインストールしたことがあり、インストーラの出力がない場合は、ソフトウェアがシステムサービスとしてインストールされているかどうかを確認できます。

AWS IoT Greengrass Core ソフトウェアがシステムサービスとしてインストールされているかどうかを確認するには

- 次のコマンドを実行して、Greengrass システムサービスの状態を確認します。

  Linux or Unix (systemd)

  ```
  sudo systemctl status greengrass.service
  ```

  AWS IoT Greengrass Core ソフトウェアがシステムサービスとしてインストールされ、アクティブである場合、応答は次の例のようになります。

  ```
  # greengrass.service - Greengrass Core
  ```

```
   Loaded: loaded (/etc/systemd/system/greengrass.service; enabled; vendor
 preset: disabled)
   Active: active (running) since Thu 2021-02-11 01:33:44 UTC; 4 days ago
 Main PID: 16107 (sh)
   CGroup: /system.slice/greengrass.service
           ##16107 /bin/sh /greengrass/v2/alts/current/distro/bin/loader
           ##16111 java -Dlog.store=FILE -Droot=/greengrass/v2 -jar /greengrass/
v2/alts/current/distro/lib/Greengrass...
```

`systemctl` または `greengrass.service` が見つからない場合、AWS IoT Greengrass Core ソフトウェアはシステムサービスとしてインストールされません。ソフトウェアを実行するには、「システムサービスを使用せずに AWS IoT Greengrass Core ソフトウェアを実行します。」を参照してください。

Windows Command Prompt (CMD)

```
sc query greengrass
```

AWS IoT Greengrass Core ソフトウェアが Windows サービスとしてインストールされ、アクティブである場合、応答は次の例のようになります。

```
SERVICE_NAME: greengrass
        TYPE               : 10  WIN32_OWN_PROCESS
        STATE              : 4  RUNNING
                                (STOPPABLE, NOT_PAUSABLE, ACCEPTS_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0
```

PowerShell

```
Get-Service greengrass
```

AWS IoT Greengrass Core ソフトウェアが Windows サービスとしてインストールされ、アクティブである場合、応答は次の例のようになります。

```
Status   Name               DisplayName
------   ----               -----------
Running  greengrass         greengrass
```

AWS IoT Greengrass Core ソフトウェアをシステムサービスとして実行する

AWS IoT Greengrass Core ソフトウェアがシステムサービスとしてインストールされている場合は、システムサービスマネージャーを使用して、ソフトウェアを開始、停止、および管理できます。詳細については、「Greengrass nucleus をシステムサービスとして設定する」を参照してください。

AWS IoT Greengrass Core ソフトウェアを実行するには

- 次のコマンドを使用して AWS IoT Greengrass Core ソフトウェアを起動します。

  Linux or Unix (systemd)

  ```
  sudo systemctl start greengrass.service
  ```

  Windows Command Prompt (CMD)

  ```
  sc start greengrass
  ```

  PowerShell

  ```
  Start-Service greengrass
  ```

システムサービスを使用せずに AWS IoT Greengrass Core ソフトウェアを実行します。

Linux コアデバイスでは、AWS IoT Greengrass Core ソフトウェアがシステムサービスとしてインストールされていない場合は、ソフトウェアのローダースクリプトを実行してソフトウェアを実行できます。

システムサービスを使用せずに AWS IoT Greengrass Core ソフトウェアを実行するには

- 次のコマンドを使用して AWS IoT Greengrass Core ソフトウェアを起動します。このコマンドをターミナルで実行する場合、ターミナルセッションを開いたままにして、AWS IoT Greengrass Core ソフトウェアを実行し続ける必要があります。

- */greengrass/v2* または *C:\greengrass\v2* を、使用する Greengrass ルートフォルダに置き換えます。

```
sudo /greengrass/v2/alts/current/distro/bin/loader
```

正常に起動すると、ソフトウェアは次のメッセージを出力します。

```
Launched Nucleus successfully.
```

Docker コンテナで AWS IoT Greengrass Core ソフトウェアを実行する

AWS IoT Greengrass は、Docker コンテナで実行するように設定できます。Docker は、Linux コンテナに基づくアプリケーションの構築、実行、テスト、およびデプロイを行うためのツールを提供するプラットフォームです。 AWS IoT Greengrass Docker イメージを実行するときに、Docker コンテナに AWS 認証情報を提供するかどうかを選択し、 AWS IoT Greengrass Core ソフトウェアインストーラが Greengrass コアデバイスの動作に必要なリソースを自動的にプロビジョニング AWS できるようにします。認証情報を提供 AWS しない場合は、 AWS リソースを手動でプロビジョニングし、Docker コンテナで AWS IoT Greengrass Core ソフトウェアを実行できます。

トピック

サポートされているプラットフォームと要件

Docker コンテナに AWS IoT Greengrass Core ソフトウェアをインストールして実行するには、ホストコンピュータが次の最小要件を満たしている必要があります。

- インターネットに接続された Linux ベースのオペレーティングシステム。
- Docker Engine バージョン 18.09 以降。
- (オプション) Docker Compose バージョン 1.22 以降。Docker Compose は、Docker Compose CLI を使用して Docker イメージを実行する場合のみ必要です。

Docker コンテナ内で Lambda 関数コンポーネントを実行するには、追加の要件を満たすようにコンテナを設定する必要があります。詳細については、「Lambda 関数の要件」を参照してください。

プロセスモードでコンポーネントを実行する

AWS IoT Greengrass は、 AWS IoT Greengrass Docker コンテナ内の分離されたランタイム環境での Lambda 関数または AWSが提供するコンポーネントの実行をサポートしていません。これらのコンポーネントは、分離せずにプロセスモードで実行する必要があります。

Lambda 関数コンポーネントを設定するときは、分離モードを [No container] (コンテナなし) に設定します。詳細については、「AWS Lambda 関数を実行する」を参照してください。

以下の AWSが提供するコンポーネントのいずれかをデプロイするときは、各コンポーネントの設定を更新して、 `containerMode`パラメータを に設定しますNoContainer。設定の更新の詳細については、「コンポーネント設定の更新」を参照してください。

- CloudWatch メトリクス
- Device Defender
- Firehose
- Modbus-RTU プロトコルアダプタ
- Amazon SNS

AWS IoT Greengrass Docker ソフトウェアのダウンロード

AWS IoT Greengrass は、Amazon Linux 2 (x86_64) ベースイメージに AWS IoT Greengrass Core ソフトウェアと依存関係がインストールされているコンテナイメージを構築するための Dockerfile を提供します。Dockerfile のベースイメージを変更して、別のプラットフォームアーキテクチャ AWS IoT Greengrass で実行できます。

から Dockerfile パッケージをダウンロードしますGitHub。

Dockerfile は古いバージョンの Greengrass を使用しています。必要なバージョンの Greengrass を使用するようにファイルを更新する必要があります。Dockerfile から AWS IoT Greengrass コンテナ

イメージを構築する方法については、「」を参照してくださいDockerfile から AWS IoT Greengrass コンテナイメージを構築する。

AWS リソースをプロビジョニングする方法を選択する

AWS IoT Greengrass Core ソフトウェアを Docker コンテナにインストールする場合、Greengrass コアデバイスの動作に必要な AWS リソースを自動的にプロビジョニングするか、手動でプロビジョニングしたリソースを使用するかを選択できます。

- 自動リソースプロビジョニング — インストーラは、 AWS IoT Greengrass コンテナイメージを初めて実行するときに、 AWS IoT モノ、 AWS IoT モノのグループ、IAM ロール、および AWS IoT ロールエイリアスをプロビジョニングします。インストーラは、ローカル開発ツールをコアデバイスにデプロイすることもできるため、デバイスを使用してカスタムソフトウェアコンポーネントを開発とテストできます。これらのリソースを自動的にプロビジョニングするには、Docker イメージに環境変数として AWS 認証情報を提供する必要があります。

  自動プロビジョニングを使用するには、Docker 環境変数 `PROVISION=true` を設定し、認証ファイルをマウントして、 AWS 認証情報をコンテナに提供する必要があります。
- 手動リソースプロビジョニング — コンテナに AWS 認証情報を提供しない場合は、 AWS IoT Greengrass コンテナイメージを実行する前にリソースを手動でプロビジョニング AWS できます。これらのリソースに関する情報を Docker コンテナ内の AWS IoT Greengrass Core ソフトウェアインストーラに提供するための設定ファイルを作成する必要があります。

  手動プロビジョニングを使用するには、Docker 環境変数 `PROVISION=false` を設定する必要があります。手動プロビジョニングがデフォルトのオプションです。

詳細については、「Dockerfile から AWS IoT Greengrass コンテナイメージを構築する」を参照してください。

Dockerfile から AWS IoT Greengrass コンテナイメージを構築する

AWS には、Docker コンテナ内での AWS IoT Greengrass Core ソフトウェアの実行を容易にするためにダウンロードして使用することができる Dockerfile が用意されています。Dockerfile には、AWS IoT Greengrass コンテナイメージを構築するためのソースコードが含まれています。

AWS IoT Greengrass コンテナイメージを構築する前に、インストールする AWS IoT Greengrass Core ソフトウェアのバージョンを選択するように Dockerfile を設定する必要があります。また、環境変数を設定することで、インストール時のリソースのプロビジョニング方法を選択したり、他の

インストールオプションをカスタマイズすることもできます。このセクションでは、Dockerfile から AWS IoT Greengrass Docker イメージを設定して構築する方法について説明します。

Dockerfile パッケージをダウンロードする

Dockerfile AWS IoT Greengrass パッケージは からダウンロードできます GitHub。

AWS Greengrass Docker リポジトリ

パッケージをダウンロードしたら、コンテンツをコンピュータ上の *download-directory*/aws-greengrass-docker-*nucleus-version* フォルダに抽出します。Dockerfile は古いバージョンの Greengrass を使用しています。必要なバージョンの Greengrass を使用するようにファイルを更新する必要があります。

AWS IoT Greengrass Core ソフトウェアのバージョンを指定する

次のビルド引数を Dockerfile で使用して、AWS IoT Greengrass Docker イメージで使用する AWS IoT Greengrass Core ソフトウェアのバージョンを指定します。デフォルトでは、Dockerfile は AWS IoT Greengrass Core ソフトウェアの最新バージョンを使用します。

`GREENGRASS_RELEASE_VERSION`

AWS IoT Greengrass Core ソフトウェアのバージョン。デフォルトでは、Dockerfile は Greengrass nucleus の利用可能な最新バージョンをダウンロードします。この値を、ダウンロードする nucleus のバージョンに設定します。

環境変数を設定する

環境変数を使用すると、AWS IoT Greengrass Core ソフトウェアが Docker コンテナにインストールされる方法をカスタマイズできます。AWS IoT Greengrass Docker イメージの環境変数は、さまざまな方法で設定できます。

- 同じ環境変数を使用して複数のイメージを作成するには、Dockerfile で環境変数を直接設定します。
- `docker run` を使用してコンテナを開始するには、コマンドで環境変数を引数として渡すか、環境変数ファイルで環境変数を設定してから、そのファイルを引数として渡します。Docker で環境変数を設定するための詳細については、Docker ドキュメントの「environment variables」を参照してください。

- `docker-compose up` を使用してコンテナを開始するには、環境変数ファイルで環境変数を設定してから、そのファイルを引数として渡します。Compose で環境変数を設定するための詳細については、「Docker ドキュメント」を参照してください。

AWS IoT Greengrass Docker イメージでは、次の環境変数を設定することができます。

> **Note**
>
> Dockerfile の `TINI_KILL_PROCESS_GROUP` 変数は変更しないでください。この変数は、Docker コンテナの停止時に、AWS IoT Greengrass Core ソフトウェアが正しくシャットダウンできるように、PID グループ内のすべての PID に `SIGTERM` を転送できるようにするためのものです。

`GGC_ROOT_PATH`

(オプション) AWS IoT Greengrass Core ソフトウェアのルートとして使用するコンテナ内のフォルダへのパス。

デフォルト: `/greengrass/v2`

`PROVISION`

(オプション) AWS IoT Greengrass Core が AWS リソースをプロビジョニングするかどうかを決定します。

- `true` を指定すると、AWS IoT Greengrass Core ソフトウェアがコンテナイメージを AWS IoT モノとして登録し、Greengrass コアデバイスが必要とする AWS リソースをプロビジョニングします。AWS IoT Greengrass Core ソフトウェアは、AWS IoT モノ、(オプションの) AWS IoT モノグループ、IAM ロール、および AWS IoT ロールエイリアスをプロビジョニングします。詳細については、「自動リソースプロビジョニングを使用して Docker コンテナで AWS IoT Greengrass を実行する」を参照してください。
- `false` を指定した場合、手動で作成した AWS リソースと証明書を使用するように指定する設定ファイルを作成して、AWS IoT Greengrass Core インストーラに提供する必要があります。詳細については、「手動リソースプロビジョニングを使用して Docker コンテナで AWS IoT Greengrass を実行する」を参照してください。

デフォルト: `false`

`AWS_REGION`

(オプション) AWS IoT Greengrass Core ソフトウェアが必要な AWS リソースを取得または作成するために使用する AWS リージョン。

デフォルト: `us-east-1`。

`THING_NAME`

(オプション) このコアデバイスとして登録する AWS IoT モノの名前。この名前を持つモノが AWS アカウント の中に存在しない場合は、AWS IoT Greengrass Core ソフトウェアが作成します。

この引数を適用するには、`PROVISION=true` を指定する必要があります。

デフォルト: `GreengrassV2IotThing_` とランダムな UUID。

`THING_GROUP_NAME`

(オプション) このコアデバイスの AWS IoT を追加した AWS IoT モノグループの名前。デプロイがこのモノグループをターゲットにしている場合、このコアデバイスとそのグループ内の他のコアデバイスは、AWS IoT Greengrass に接続するときにそのデプロイを受信します。この名前のモノグループが AWS アカウント に存在しない場合、AWS IoT Greengrass Core ソフトウェアが作成します。

この引数を適用するには、`PROVISION=true` を指定する必要があります。

`TES_ROLE_NAME`

(オプション) Greengrass コアデバイスが AWS サービスとやり取りできるようにする AWS 認証情報を取得するために使用する IAM ロールの名前。この名前のロールが AWS アカウント に存在しない場合、AWS IoT Greengrass Core ソフトウェアが `GreengrassV2TokenExchangeRoleAccess` ポリシーを使ってロールを作成します。このロールは、コンポーネントのアーティファクトをホストする S3 バケットにはアクセスできません。そのため、コンポーネントを作成するときに、アーティファクトの S3 バケットとオブジェクトへのアクセス許可を追加する必要があります。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

デフォルト: `GreengrassV2TokenExchangeRole`

`TES_ROLE_ALIAS_NAME`

(オプション) Greengrass コアデバイスの AWS 認証情報を提供する IAM ロールをポイントしている AWS IoT ロールエイリアスの名前。この名前のロールエイリアスが AWS アカウント に存

在しない場合、AWS IoT Greengrass Core ソフトウェアはそれを作成し、指定されている IAM ロールを指すように設定します。

デフォルト: `GreengrassV2TokenExchangeRoleAlias`

`COMPONENT_DEFAULT_USER`

(オプション) AWS IoT Greengrass Core ソフトウェアがコンポーネントを実行するために使用するシステムユーザーとグループの名前または ID。ユーザーとグループは、コロンで区切って指定します。グループはオプションです。たとえば、**`ggc_user:ggc_group`** や **`ggc_user`** と指定することができます。

- root として実行した場合、設定ファイルで定義したユーザーとグループがデフォルトとなります。設定ファイルでユーザーとグループが定義されていない場合、デフォルトは `ggc_user:ggc_group` になります。もし `ggc_user` または `ggc_group` が存在しない場合は、ソフトウェアが作成します。
- 非ルートユーザーとして実行すると、AWS IoT Greengrass Core ソフトウェアはそのユーザーとしてコンポーネントを実行します。
- グループを指定しなかった場合、AWS IoT Greengrass Core ソフトウェアは、システムユーザーのプライマリグループを使用します。

詳細については、「コンポーネントを実行するユーザーを設定する」を参照してください。

`DEPLOY_DEV_TOOLS`

Greengrass CLI コンポーネントをコンテナイメージにダウンロードし、デプロイするかどうかを定義します。Greengrass CLI を使用して、コンポーネントをローカルで開発およびデバッグすることができます。

> **Important**
>
> このコンポーネントは、本番環境ではなく、開発環境でのみで使用することをお勧めします。このコンポーネントは、通常、本番環境では必要とされない情報や操作へのアクセスを提供します。このコンポーネントを必要なコアデバイスにのみデプロイして、最小特権の原則に従います。

デフォルト: `false`

`INIT_CONFIG`

(オプション) AWS IoT Greengrass Core ソフトウェアのインストールに使用する設定ファイルへのパス。このオプションを使用して、特定の nucleus 設定で新しい Greengrass コアデバイスをセットアップしたり、手動でプロビジョニングされたリソースを指定したりできます。設定ファイルは、この引数で指定したパスにマウントする必要があります。

`TRUSTED_PLUGIN`

この機能は、Greengrass nucleus コンポーネントの v2.4.0 以降に利用できます。

(オプション) 信頼されたプラグインとして読み込む JAR ファイルへのパス。フリートプロビジョニングやカスタムプロビジョニングでインストールする場合など、プロビジョニングプラグインの JAR ファイルを提供する場合にこのオプションを使用します。

`THING_POLICY_NAME`

この機能は、Greengrass nucleus コンポーネントの v2.4.0 以降に利用できます。

(オプション) このコアデバイスの AWS IoT モノ証明書にアタッチする AWS IoT ポリシーの名前。この名前の AWS IoT ポリシーが AWS アカウント に存在しない場合は、AWS IoT Greengrass Core ソフトウェアが作成します。

この引数を適用するには、`PROVISION=true` を指定する必要があります。

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアでは、デフォルトで寛容な AWS IoT ポリシーが作成されます。このポリシーの範囲を絞り込むか、ユースケースに応じた権限に制限するカスタムポリシーを作成することができます。詳細については、「AWS IoT Greengrass V2 コアデバイス向けの最低限の AWS IoT ポリシー」を参照してください。

インストールする依存関係を指定する

AWS IoT Greengrass Dockerfile の RUN 命令は、AWS IoT Greengrass Core ソフトウェアインストーラを実行するためのコンテナ環境を準備します。Docker コンテナで AWS IoT Greengrass Core ソフトウェアのインストーラを実行する前に、インストールされる依存関係をカスタマイズできます。

AWS IoT Greengrass イメージを構築する

AWS IoT Greengrass Dockerfile を使用して AWS IoT Greengrass コンテナイメージを構築する Docker CLI または Docker Compose CLI を使用してイメージを構築し、コンテナを開始することができます。また、Docker CLI を使用してイメージを構築し、その後 Docker Compose を使用してそのイメージからコンテナを開始することもできます。

Docker

1. ホストマシンで次のコマンドを実行し、設定した Dockerfile が含まれるディレクトリに切り替えます。

   ```
   cd download-directory/aws-greengrass-docker-nucleus-version
   ```

2. 次のコマンドを実行して、Dockerfile から AWS IoT Greengrass コンテナイメージを構築します。

   ```
   sudo docker build -t "platform/aws-iot-greengrass:nucleus-version" ./
   ```

Docker Compose

1. ホストマシンで次のコマンドを実行し、Dockerfile ファイルと Compose ファイルが含まれてるディレクトリに切り替えます。

   ```
   cd download-directory/aws-greengrass-docker-nucleus-version
   ```

2. 次のコマンドを実行して、Compose ファイルを使用して AWS IoT Greengrass コンテナイメージを構築します。

   ```
   docker-compose -f docker-compose.yml build
   ```

AWS IoT Greengrass コンテナイメージが作成できました。Docker イメージには、AWS IoT Greengrass Core ソフトウェアがインストールされています。これで、Docker コンテナで AWS IoT Greengrass Core ソフトウェアを実行できるようになりました。

自動リソースプロビジョニングを使用して Docker コンテナで AWS IoT Greengrass を実行する

このチュートリアルでは、自動的にプロビジョニングされた AWS リソースとローカル開発ツールを使用して Docker コンテナに AWS IoT Greengrass Core ソフトウェアをインストールして実行する方法を示します。この開発環境を使用して、Docker コンテナの AWS IoT Greengrass 機能を調べることができます。ソフトウェアには、これらのリソースをプロビジョニングし、ローカル開発ツールをデプロイするための AWS 認証情報が必要です。

コンテナに AWS 認証情報を提供できない場合、コアデバイスの動作に必要な AWS リソースをプロビジョニングできます。開発ツールをコアデバイスにデプロイして、開発デバイスとして使用することもできます。これにより、コンテナの実行時にデバイスに付与するアクセス許可を減らすことができます。詳細については、「手動リソースプロビジョニングを使用して Docker コンテナで AWS IoT Greengrass を実行する」を参照してください。

前提条件

このチュートリアルを完了するには、以下が必要です。

- AWS アカウント。アカウントをお持ちでない場合は、「AWS アカウント のセットアップ」を参照してください。
- Greengrass コアデバイスの AWS IoT および IAM リソースをプロビジョニングする権限を持つ AWS IAM ユーザー。AWS IoT Greengrass Core ソフトウェアインストーラは、お客様の AWS 認証情報を使用して、これらのリソースを自動的にプロビジョニングします。リソースを自動的にプロビジョニングする最小の IAM ポリシーの詳細については、「インストーラがリソースをプロビジョニングするための最小限の IAM ポリシー」を参照してください。
- AWS IoT Greengrass Docker イメージ。AWS IoT Greengrass Dockerfile からイメージを構築できます。
- Docker コンテナを実行するホストコンピュータは、以下の要件を満たしている必要があります。
  - インターネットに接続された Linux ベースのオペレーティングシステム。
  - Docker Engine バージョン 18.09 以降。
  - (オプション) Docker Compose バージョン 1.22 以降。Docker Compose は、Docker Compose CLI を使用して Docker イメージを実行する場合のみ必要です。

AWS 認証情報を設定する

このステップでは、AWS セキュリティ認証情報を含む認証情報ファイルをホストコンピュータに作成します。AWS IoT Greengrass Dockerイメージを実行するときは、この認証情報ファイルを含むフォルダをDocker コンテナ内の `/root/.aws/` にマウントする必要があります。AWS IoT Greengrass インストーラは、これらの認証情報を使用して、お客様の AWS アカウント のリソースをプロビジョニングします。インストーラがリソースを自動的にプロビジョニングするために必要な最小 IAM ポリシーについては、「インストーラがリソースをプロビジョニングするための最小限の IAM ポリシー」を参照してください。

1. 次のいずれかを取得します。

   - IAM ユーザーの長期的な認証情報。長期認証情報を取得する方法については、「IAM ユーザーガイド」の「[Managing access keys for IAM users] (IAM ユーザーのアクセスキー管理)」を参照してください。
   - (推奨) IAM ロールの一時的な認証情報。一時的な認証情報を取得する方法については、「IAM ユーザーガイド」の「AWS CLI で一時的なセキュリティ認証情報を使用する」を参照してください。

2. 認証情報ファイルを配置するフォルダを作成します。

   ```
   mkdir ./greengrass-v2-credentials
   ```

3. テキストエディタを使用して、`./greengrass-v2-credentials` フォルダに `credentials` という名前の設定ファイルを作成します。

   例えば、次のコマンドを実行し、GNU nano を使用して `credentials` ファイルを作成できます。

   ```
   nano ./greengrass-v2-credentials/credentials
   ```

4. AWS 認証情報を次の形式で `credentials` ファイルに追加します。

   ```
   [default]
   aws_access_key_id = AKIAIOSFODNN7EXAMPLE
   aws_secret_access_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
   aws_session_token
    = AQoEXAMPLEH4aoAH0gNCAPy...truncated...zrkuWJOgQs8IZZaIv2BXIa2R4Olgk
   ```

   一時的な認証情報の場合のみ `aws_session_token` を含めます。

> **Important**
>
> AWS IoT Greengrass コンテナを起動した後、ホストコンピュータから認証情報ファイルを削除します。認証情報ファイルを削除しない場合は、AWS 認証情報はコンテナ内にマウントされたままになります。詳細については、「コンテナで AWS IoT Greengrass Core ソフトウェアを実行する」を参照してください。

環境ファイルを作成する

このチュートリアルでは、環境ファイルを使用して、Docker コンテナ内の AWS IoT Greengrass Core ソフトウェアインストーラに渡される環境変数を設定します。また、`docker run` コマンドで `-e または --env 引数`を使用して、Docker コンテナに環境変数を設定する、または `docker-compose.yml` ファイルの `environment ブロック`で変数を設定することもできます。

1. テキストエディタを使用して、`.env` という名前の環境ファイルを作成します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用して現在のディレクトリに `.env` を作成できます。

   ```
   nano .env
   ```

2. 次の内容をファイルにコピーします。

   ```
   GGC_ROOT_PATH=/greengrass/v2
   AWS_REGION=region
   PROVISION=true
   THING_NAME=MyGreengrassCore
   THING_GROUP_NAME=MyGreengrassCoreGroup
   TES_ROLE_NAME=GreengrassV2TokenExchangeRole
   TES_ROLE_ALIAS_NAME=GreengrassCoreTokenExchangeRoleAlias
   COMPONENT_DEFAULT_USER=ggc_user:ggc_group
   ```

   次に、以下の値を置き換えます。

   - */greengrass/v2*。 インストールに使用する Greengrass ルートフォルダ。GGC_ROOT 環境変数を使用して、この値を設定します。
   - *#####*。リソースを作成した AWS リージョン 。

- *MyGreengrassCore*。AWS IoT モノの名前。モノが存在しない場合、インストーラによって作成されます。インストーラは、証明書をダウンロードして AWS IoT モノとして認証します。
- *MyGreengrassCoreGroup*。AWS IoT モノのグループの名前。モノグループが存在しない場合、インストーラはそのグループを作成してモノを追加します。モノグループが存在してアクティブなデプロイがある場合、コアデバイスはデプロイで指定されたソフトウェアをダウンロードして実行します。
- *GreengrassV2TokenExchangeRole*。Greengrass コアデバイスが一時的な AWS 認証情報を取得できるようにする IAM トークン交換ロールの名前に置き換えます。ロールが存在しない場合、インストーラはロールを作成し、*GreengrassV2TokenExchangeRole*Access という名前のポリシーを作成してアタッチします。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。
- *GreengrassCoreTokenExchangeRoleAlias*。トークン交換ロールエイリアス。ロールエイリアスが存在しない場合、インストーラがロールエイリアスを作成し、指定したIAMトークン交換ロールを指します。詳細については、以下を参照してください。

> **Note**
>
> DEPLOY_DEV_TOOLS 環境変数を true に設定して、[Greengrass CLI component] (Greengrass CLI コンポーネント) をデプロイできます。これにより、Docker コンテナ内でカスタムコンポーネントを開発できます。このコンポーネントは、本番環境ではなく、開発環境でのみで使用することをお勧めします。このコンポーネントは、通常、本番環境では必要とされない情報や操作へのアクセスを提供します。このコンポーネントを必要なコアデバイスにのみデプロイして、最小特権の原則に従います。

コンテナで AWS IoT Greengrass Core ソフトウェアを実行する

このチュートリアルでは、Docker コンテナでビルドした Docker イメージを起動する方法を説明します。Docker CLI または Docker Compose CLI を使用して、Docker コンテナ内の AWS IoT Greengrass Core ソフトウェアイメージを実行できます。

Docker

1. 次のコマンドを実行して Docker コンテナを起動します。

```
docker run --rm --init -it --name docker-image \
 -v path/to/greengrass-v2-credentials:/root/.aws/:ro \
 --env-file .env \
 -p 8883 \
 your-container-image:version
```

このコマンド例は、docker run に次の引数を使用します。

- `--rm`: コンテナの終了時にクリーンアップを実行します。
- `--init`: コンテナで init プロセスを使用します。

> **Note**
>
> Docker コンテナを停止するときに AWS IoT Greengrass Core ソフトウェアをシャットダウンするには、`--init` 引数が必要です。

- `-it`: (オプション) Docker コンテナを対話型プロセスとしてフォアグラウンドで実行します。これを `-d` 引数に置き換えて、代わりに Docker コンテナをデタッチモードで実行できます。詳細については、「Docker ドキュメント」の「デタッチ vs フォアグラウンド」を参照してください。
- `--name`: `aws-iot-greengrass` という名前のコンテナを実行します。
- `-v`: ボリュームを Docker コンテナにマウントして、コンテナ内で実行されている AWS IoT Greengrass で設定ファイルと証明書ファイルを利用可能にします。
- `--env-file`: (オプション) Docker コンテナ内の AWS IoT Greengrass Core ソフトウェアインストーラに渡される環境変数を設定する環境ファイルを指定します。この引数は、環境変数を設定するための [environment file] (環境ファイル) を作成した場合にのみ必要です。環境ファイルを作成していない場合は、`--env` 引数を使用して、「Docker 実行コマンド」で環境変数を直接設定できます。
- `-p`: (オプション) 8883 コンテナポートをホストマシンに公開します。AWS IoT Greengrass は MQTT トラフィックにポート 8883 を使用するため、MQTT を介して接続および通信する場合は、この引数が必要です。他のポートを開くには、追加の `-p` 引数を使用します。

> **Note**
>
> セキュリティを強化して Docker コンテナを実行するには、`--cap-drop` 引数および `--cap-add` 引数を使用して、コンテナの Linux 機能を選択的に有効にします。詳細については、「Docker ドキュメント」の「[Runtime privilege and Linux capabilities] (ランタイム特権と Linux 機能)」を参照してください。

2. ホストデバイスで `./greengrass-v2-credentials` から認証情報を削除します。

```
rm -rf ./greengrass-v2-credentials
```

> **Important**
>
> これらの認証情報は、コアデバイスがセットアップ時にのみ必要とする広範なアクセス許可を提供するため、削除します。これらの認証情報を削除しないと、Greengrass コンポーネントとコンテナで実行されている他のプロセスがそれらにアクセスできます。AWS 認証情報を Greengrass コンポーネントに提供する必要がある場合は、トークン交換サービスを使用します。詳細については、「AWS サービスとやり取り」を参照してください。

Docker Compose

1. テキストエディタを使用して、`docker-compose.yml` という名前の Docker Compose ファイルを作成します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用して現在のディレクトリに `docker-compose.yml` を作成できます。

```
nano docker-compose.yml
```

> **Note**
>
> から AWSが提供する Compose ファイルの最新バージョンをダウンロードして使用することもできますGitHub。

2. Compose ファイルに以下の内容を追加します。ファイルは次の例のようになります。*docker-image* をお使いの Docker イメージの名前に置き換えます。

```
version: '3.7'

services:
  greengrass:
    init: true
    container_name: aws-iot-greengrass
    image: docker-image
    volumes:
      - ./greengrass-v2-credentials:/root/.aws/:ro
    env_file: .env
    ports:
      - "8883:8883"
```

この例の Compose ファイルでは、以下のパラメータはオプションです。

- `ports` - 8883 コンテナポートをホストマシンに公開します。AWS IoT Greengrass は MQTT トラフィックにポート 8883 を使用するため、MQTT を介して接続および通信する場合は、このパラメータが必要です。
- `env_file` - Docker コンテナ内の AWS IoT Greengrass Core ソフトウェアインストーラに渡される環境変数を設定する環境ファイルを指定します。このパラメータは、環境変数を設定するための [environment file] (環境ファイル) を作成した場合にのみ必要です。環境ファイルを作成していない場合は、[environment] (環境) パラメータを使用して、Compose ファイルで環境変数を直接設定できます。

> **Note**
>
> セキュリティを強化して Docker コンテナを実行するには、Compose ファイルで `cap_drop` および `cap_add` を使用して、コンテナの Linux 機能を選択的に有効にします。詳細については、「Docker ドキュメント」の「[Runtime privilege and Linux capabilities] (ランタイム特権と Linux 機能)」を参照してください。

3. 次のコマンドを実行して Docker コンテナを起動します。

```
docker-compose -f docker-compose.yml up
```

4. ホストデバイスで `./greengrass-v2-credentials` から認証情報を削除します。

```
rm -rf ./greengrass-v2-credentials
```

> **Important**
>
> これらの認証情報は、コアデバイスがセットアップ時にのみ必要とする広範なアクセス許可を提供するため、削除します。これらの認証情報を削除しないと、Greengrass コンポーネントとコンテナで実行されている他のプロセスがそれらにアクセスできます。AWS 認証情報を Greengrass コンポーネントに提供する必要がある場合は、トークン交換サービスを使用します。詳細については、「AWS サービスとやり取り」を参照してください。

次のステップ

現在、Docker コンテナで AWS IoT Greengrass Core ソフトウェアが実行されています。次のコマンドを実行して、現在実行中のコンテナのコンテナ ID を取得します。

```
docker ps
```

次に、次のコマンドを実行してコンテナにアクセスし、コンテナ内で実行されている AWS IoT Greengrass Core ソフトウェアを検索できます。

```
docker exec -it container-id /bin/bash
```

単純なコンポーネントの作成については、「チュートリアル: AWS IoT Greengrass V2 の開始方法」の「ステップ 4: デバイス上でコンポーネントを開発およびテストする」を参照してください。

> **Note**
>
> `docker exec` を使用して Docker コンテナ内でコマンドを実行すると、これらのコマンドは Docker ログに記録されません。Docker ログにコマンドを記録するには、Docker コンテナに対話型シェルをアタッチします。詳細については、「インタラクティブシェルを Docker コンテナにアタッチする」を参照してください。

AWS IoT Greengrass Core ログファイルは `greengrass.log` と呼ばれ、*/greengrass/v2*/logs にあります。コンポーネントログファイルも同じディレクトリにあります。Greengrass ログをホストの一時ディレクトリにコピーするには、次のコマンドを実行します。

```
docker cp container-id:/greengrass/v2/logs /tmp/logs
```

コンテナの終了後、または削除後もログを保持する場合は、Greengrassディレクトリ全体をマウントするのではなく、*/greengrass/v2*/logs ディレクトリのみをホストの一時ログディレクトリにバインドマウントすることをお勧めします。詳細については、「Docker コンテナの外部で Greengrass ログを永続化する」を参照してください。

実行中の AWS IoT Greengrass Docker コンテナを停止するには、`docker stop` または `docker-compose -f docker-compose.yml stop` を実行します。このアクションは、`SIGTERM` を Greengrass プロセスに送信し、コンテナで開始されたすべての関連プロセスをシャットダウンします。Docker コンテナは、プロセス PID 1 として `docker-init` の実行可能ファイルで初期化されます。これは、残っているゾンビプロセスを削除するのに役立ちます。詳細については、Docker ドキュメントの「Specify an init process」を参照してください。

Docker コンテナで AWS IoT Greengrass を実行する際の問題のトラブルシューティングについては、「Docker コンテナでの AWS IoT Greengrass のトラブルシューティング」を参照してください。

手動リソースプロビジョニングを使用して Docker コンテナで AWS IoT Greengrass を実行する

このチュートリアルでは、手動でプロビジョニングされた AWS リソースを使用して Docker コンテナに AWS IoT Greengrass Core ソフトウェアをインストールして実行する方法を示します。

トピック

- 前提条件
- AWS IoT エンドポイントの取得
- AWS IoT のモノを作成する
- モノの証明書を作成する
- モノの証明書を設定する
- トークン交換ロールを作成する
- デバイスに証明書をダウンロードする
- 設定ファイルを作成する

前提条件

このチュートリアルを完了するには、以下が必要です。

- AWS アカウント。アカウントをお持ちでない場合は、「AWS アカウント のセットアップ」を参照してください。
- AWS IoT Greengrass Docker イメージ。AWS IoT Greengrass Dockerfile からイメージを構築できます。
- Docker コンテナを実行するホストコンピュータは、以下の要件を満たしている必要があります。
  - インターネットに接続された Linux ベースのオペレーティングシステム。
  - Docker Engine バージョン 18.09 以降。
  - (オプション) Docker Compose バージョン 1.22 以降。Docker Compose は、Docker Compose CLI を使用して Docker イメージを実行する場合のみ必要です。

AWS IoT エンドポイントの取得

AWS アカウント に AWS IoT エンドポイントを入手して、後で使用するために保存してください。デバイスはこれらのエンドポイントを使用して AWS IoT に接続します。以下の操作を実行します。

1. AWS アカウント の AWS IoT データエンドポイントを取得します。

   ```
   aws iot describe-endpoint --endpoint-type iot:Data-ATS
   ```

   要求が正常に処理された場合、レスポンスは次の例のようになります。

   ```
   {
     "endpointAddress": "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
   }
   ```

2. AWS IoT の AWS アカウント 認証情報エンドポイントを取得します。

   ```
   aws iot describe-endpoint --endpoint-type iot:CredentialProvider
   ```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "endpointAddress": "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
}
```

AWS IoT のモノを作成する

AWS IoT モノは AWS IoT に接続するデバイスと論理エンティティを表します。Greengrass コアデバイスは AWS IoT モノです。デバイスを AWS IoT モノとして登録するとき、そのデバイスはデジタル証明書を使用して AWS で認証できます。

このセクションでは、デバイスを表す AWS IoT モノを作成します。

AWS IoT モノを作成するには

1. デバイスの AWS IoT モノを作成します。開発用コンピュータに次のコマンドを実行します。

   - を使用するモノの名前*MyGreengrassCore*に置き換えます。この名前は Greengrass コアデバイスの名前でもあります。

     > **Note**
     >
     > モノの名前にコロン (:) 記号を含むことができません。

```
aws iot create-thing --thing-name MyGreengrassCore
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "thingName": "MyGreengrassCore",
  "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
  "thingId": "8cb4b6cd-268e-495d-b5b9-1713d71dbf42"
}
```

2. (オプション) AWS IoT モノを新規または既存のモノグループに追加します。モノグループを使用して Greengrass コアデバイスのフリートを管理します。ソフトウェアコンポーネントをデバイスにデプロイするとき、個々のデバイスまたはデバイスのグループを対象にできます。アクティブな Greengrass デプロイを持つモノグループにデバイスを追加して、そのモノグループのソフトウェアコンポーネントをデバイスにデプロイできます。以下の操作を実行します。

   a. (オプション) AWS IoT モノグループを作成します。

      - を、作成するモノのグループの名前*MyGreengrassCoreGroup*に置き換えます。

      > **Note**
      >
      > モノグループ名にコロン (:) 記号を含めることはできません。

      ```
      aws iot create-thing-group --thing-group-name MyGreengrassCoreGroup
      ```

      要求が正常に処理された場合、レスポンスは次の例のようになります。

      ```
      {
        "thingGroupName": "MyGreengrassCoreGroup",
        "thingGroupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/
      MyGreengrassCoreGroup",
        "thingGroupId": "4df721e1-ff9f-4f97-92dd-02db4e3f03aa"
      }
      ```

   b. AWS IoT モノをモノグループに追加します。

      - をAWS IoTモノの名前*MyGreengrassCore*に置き換えます。
      - をモノのグループの名前*MyGreengrassCoreGroup*に置き換えます。

      ```
      aws iot add-thing-to-thing-group --thing-name MyGreengrassCore --thing-group-
      name MyGreengrassCoreGroup
      ```

      要求が正常に処理された場合、コマンドは出力されません。

モノの証明書を作成する

デバイスを AWS IoT モノとして登録するとき、そのデバイスはデジタル証明書を使用して AWS で認証できます。この証明書は、デバイスが AWS IoT と AWS IoT Greengrass と通信できるようにします。

このセクションでは、デバイスが AWS に接続する際に使用できる証明書を作成してダウンロードします。

モノの証明書を作成するには

1. AWS IoT モノの証明書をダウンロードするフォルダを作成します。

```
mkdir greengrass-v2-certs
```

2. AWS IoT モノの証明書を作成してダウンロードします。

```
aws iot create-keys-and-certificate --set-as-active --certificate-pem-outfile
 greengrass-v2-certs/device.pem.crt --public-key-outfile greengrass-v2-certs/
public.pem.key --private-key-outfile greengrass-v2-certs/private.pem.key
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "certificateArn": "arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
  "certificateId":
 "aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
  "certificatePem": "-----BEGIN CERTIFICATE-----
MIICiTCCAfICCQD6m7oRw0uXOjANBgkqhkiG9w
 0BAQUFADCBiDELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZ
 WF0dGxlMQ8wDQYDVQQKEwZBbWF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIw
 EAYDVQQDEwlUZXN0Q2lsYWMxHzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5
 jb20wHhcNMTEwNDI1MjA0NTIxWhcNMTIwNDI0MjA0NTIxWjCBiDELMAkGA1UEBh
 MCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBb
 WF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIwEAYDVQQDEwlUZXN0Q2lsYWMx
 HzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5jb20wgZ8wDQYJKoZIhvcNAQE
 BBQADgY0AMIGJAoGBAMaK0dn+a4GmWIWJ21uUSfwfEvySWtC2XADZ4nB+BLYgVI
 k60CpiwsZ3G93vUEIO3IyNoH/f0wYK8m9TrDHudUZg3qX4waLG5M43q7Wgc/MbQ
 ITxOUSQv7c7ugFFDzQGBzZswY6786m86gpEIbb3OhjZnzcvQAaRHhdlQWIMm2nr
 AgMBAAEwDQYJKoZIhvcNAQEFBQADgYEAtCu4nUhVVxYUntneD9+h8Mg9q6q+auN
 KyExzyLwaxlAoo7TJHidbtS4J5iNmZgXL0FkbFFBjvSfpJIlJ00zbhNYS5f6Guo
```

```
 EDmFJl0ZxBHjJnyp378OD8uTs7fLvjx79LjSTbNYiytVbZPQUQ5Yaxu2jXnimvw
 3rrszlaEXAMPLE=
-----END CERTIFICATE-----",
  "keyPair": {
    "PublicKey": "-----BEGIN PUBLIC KEY-----\
MIIBIjANBgkqhkEXAMPLEQEFAAOCAQ8AMIIBCgKCAQEAEXAMPLE1nnyJwKSMHw4h\
MMEXAMPLEuuN/dMAS3fyce8DW/4+EXAMPLEyjmoF/YVF/gHr99VEEXAMPLE5VF13\
59VK7cEXAMPLE67GK+y+jikqXOgHh/xJTwo
+sGpWEXAMPLEDz18xOd2ka4tCzuWEXAMPLEahJbYkCPUBSU8opVkR7qkEXAMPLE1DR6sx2HocliOOLtu6Fkw91swQWE
\\GB3ZPrNh0PzQYvjUStZeccyNCx2EXAMPLEvp9mQOUXP6plfgxwKRX2fEXAMPLEDa\
hJLXkX3rHU2xbxJSq7D+XEXAMPLEcw+LyFhI5mgFRl88eGdsAEXAMPLElnI9EesG\
FQIDAQAB\
-----END PUBLIC KEY-----\
",
    "PrivateKey": "-----BEGIN RSA PRIVATE KEY-----\
key omitted for security reasons\
-----END RSA PRIVATE KEY-----\
"
  }
}
```

後で証明書を設定するために使用する証明書の Amazon リソースネーム (ARN) を保存します。

モノの証明書を設定する

モノの証明書を以前に作成した AWS IoT モノにアタッチし、AWS IoT ポリシーを証明書に追加してコアデバイスの AWS IoT 許可を定義します。

モノの証明書を設定するには

1. 証明書を AWS IoT モノにアタッチします。

   - をAWS IoTモノの名前*MyGreengrassCore*に置き換えます。
   - 証明書 Amazon リソースネーム (ARN) を、前のステップで作成した証明書の ARN に置き換えます。

```
aws iot attach-thing-principal --thing-name MyGreengrassCore
 --principal arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
```

要求が正常に処理された場合、コマンドは出力されません。

2. Greengrass コアデバイスの AWS IoT 許可を定義する AWS IoT ポリシーを作成してアタッチします。次のポリシーは、すべての MQTT トピックと Greengrass 操作へのアクセスを許可するため、デバイスがカスタムアプリケーションや新しい Greengrass 操作を必要とする今後の変更でも動作するようになります。ユースケースに基づいてこのポリシーを制限できます。詳細については、「AWS IoT Greengrass V2 コアデバイス向けの最低限の AWS IoT ポリシー」を参照してください。

   Greengrass コアデバイスを以前にセットアップしたことがある場合、新しく作成せず、その AWS IoT ポリシーをアタッチできます。

   以下の操作を実行します。

   a. Greengrass コアデバイスが必要な AWS IoT ポリシードキュメントを含むファイルを作成します。

      例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

      ```
      nano greengrass-v2-iot-policy.json
      ```

      次の JSON をファイルにコピーします。

      ```
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Action": [
              "iot:Publish",
              "iot:Subscribe",
              "iot:Receive",
              "iot:Connect",
              "greengrass:*"
            ],
            "Resource": [
              "*"
            ]
          }
        ]
      ```

```
}
```

b. ポリシードキュメントから AWS IoT ポリシーを作成します。

- *GreengrassV2IoTThingPolicy* を作成するポリシーの名前に置き換えます。

```
aws iot create-policy --policy-name GreengrassV2IoTThingPolicy --policy-
document file://greengrass-v2-iot-policy.json
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "policyName": "GreengrassV2IoTThingPolicy",
  "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy",
  "policyDocument": "{
    \\"Version\\": \\"2012-10-17\\",
    \\"Statement\\": [
      {
        \\"Effect\\": \\"Allow\\",
        \\"Action\\": [
          \\"iot:Publish\\",
          \\"iot:Subscribe\\",
          \\"iot:Receive\\",
          \\"iot:Connect\\",
          \\"greengrass:*\\"
        ],
        \\"Resource\\": [
          \\"*\\"
        ]
      }
    ]
  }",
  "policyVersionId": "1"
}
```

c. AWS IoT ポリシーを AWS IoT モノの証明書にアタッチします。

- *GreengrassV2IoTThingPolicy* をアタッチするポリシーの名前に置き換えます。
- ターゲット ARN を AWS IoT モノの証明書の ARN に置き換えます。

```
aws iot attach-policy --policy-name GreengrassV2IoTThingPolicy
 --target arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
```

要求が正常に処理された場合、コマンドは出力されません。

トークン交換ロールを作成する

Greengrass コアデバイスは、トークン交換ロールという IAM サービスロールを使用して、AWS サービスへの通話を承認します。デバイスはAWS IoT認証情報プロバイダーを使用して、このロールの一時的なAWS認証情報を取得します。これにより、デバイスは とやり取りしAWS IoT、Amazon CloudWatch Logs にログを送信し、Amazon S3 からカスタムコンポーネントアーティファクトをダウンロードできるようになります。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

AWS IoT のロールエイリアスを使用して、Greengrass コアデバイスのトークン交換ロールを設定します。ロールエイリアスは、デバイスのトークン交換ロールを変更できるようにしますが、デバイス設定は同じ内容に保たれます。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS サービスへの直接呼び出しを認証する」を参照してください。

このセクションでは、トークン交換 IAM ロールとロールを指す AWS IoT ロールエイリアスを作成します。Greengrass コアデバイスを既に設定している場合、新しく作成せず、トークン交換ロールとロールエイリアスを使用できます。次に、デバイスの AWS IoT モノを設定してそのロールとエイリアスを使用します。

トークン交換 IAM ロールを作成するには

1. デバイスがトークン交換ロールとして使用できる IAM ロールを作成します。以下の操作を実行します。

   a. トークン交換ロールが必要とする、信頼できるポリシードキュメントが含まれるファイルを作成します。

      例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

      ```
      nano device-role-trust-policy.json
      ```

次の JSON をファイルにコピーします。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.iot.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

b. 信頼ポリシードキュメントでトークン交換ロールを作成します。

- *GreengrassV2TokenExchangeRole* を作成する IAM ロールの名前に置き換えます。

```
aws iam create-role --role-name GreengrassV2TokenExchangeRole --assume-role-
policy-document file://device-role-trust-policy.json
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "Role": {
    "Path": "/",
    "RoleName": "GreengrassV2TokenExchangeRole",
    "RoleId": "AROAZ2YMUHYHK5OKM77FB",
    "Arn": "arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole",
    "CreateDate": "2021-02-06T00:13:29+00:00",
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "credentials.iot.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
```

```
      ]
    }
  }
```

c. トークン交換ロールが必要なアクセスポリシードキュメントを含むファイルを作成します。

例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano device-role-access-policy.json
```

次の JSON をファイルにコピーします。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
}
```

> **Note**
>
> このアクセスポリシーでは、S3 バケットのコンポーネントアーティファクトへのアクセスが許可されていません。Amazon S3 でアーティファクトを定義するカスタムコンポーネントをデプロイするには、コアデバイスがコンポーネントアーティファクトを取得できるようにする許可をロールに追加する必要があります。詳細については、「コンポーネントのアーティファクトの S3 バケットへのアクセスを許可する」を参照してください。
> コンポーネントアーティファクトに S3 バケットをまだ持っていない場合、バケットを作成した後でこれらのアクセス許可を追加できます。

d. ポリシードキュメントから IAM ポリシーを作成します。

- *GreengrassV2TokenExchangeRoleAccess* を作成する IAM ポリシーの名前に置き換えます。

```
aws iam create-policy --policy-name GreengrassV2TokenExchangeRoleAccess --
policy-document file://device-role-access-policy.json
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "Policy": {
    "PolicyName": "GreengrassV2TokenExchangeRoleAccess",
    "PolicyId": "ANPAZ2YMUHYHACI7C5Z66",
    "Arn": "arn:aws:iam::123456789012:policy/
GreengrassV2TokenExchangeRoleAccess",
    "Path": "/",
    "DefaultVersionId": "v1",
    "AttachmentCount": 0,
    "PermissionsBoundaryUsageCount": 0,
    "IsAttachable": true,
    "CreateDate": "2021-02-06T00:37:17+00:00",
    "UpdateDate": "2021-02-06T00:37:17+00:00"
  }
}
```

e. IAM ポリシーをトークン交換ロールにアタッチします。

- *GreengrassV2TokenExchangeRole* を IAM ロールの名前に置き換えます。
- ポリシー ARN を前のステップで作成した IAM ポリシーの ARN に置き換えます。

```
aws iam attach-role-policy --role-name GreengrassV2TokenExchangeRole --policy-
arn arn:aws:iam::123456789012:policy/GreengrassV2TokenExchangeRoleAccess
```

要求が正常に処理された場合、コマンドは出力されません。

2. トークン交換ロールを指す AWS IoT ロールエイリアスを作成します。

- を、作成するロールエイリアスの名前*GreengrassCoreTokenExchangeRoleAlias*に置き換えます。

- ロール ARN を前のステップで作成した IAM ロールの ARN に置き換えます。

```
aws iot create-role-alias --role-alias GreengrassCoreTokenExchangeRoleAlias --role-arn arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "roleAlias": "GreengrassCoreTokenExchangeRoleAlias",
  "roleAliasArn": "arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias"
}
```

> **Note**
>
> ロールエイリアスを作成するには、トークン交換 IAM ロールを AWS IoT に渡す許可が必要です。ロールエイリアスの作成時にエラーメッセージが表示された場合、AWS ユーザーがこの許可を得ていることを確認てください。詳細については、「AWS Identity and Access Management ユーザーガイド」の「AWS サービスにロールをわたす許可をユーザーに付与する」を参照してください。

3. Greengrass コアデバイスがロールエイリアスを使用して、トークン交換ロールを引き受けることを許可する AWS IoT ポリシーを作成とアタッチします。Greengrass コアデバイスをセットアップしたことがある場合、新しく作成せず、そのロールエイリアスの AWS IoT ポリシーをアタッチできます。以下の操作を実行します。

   a. (オプション) ロールエイリアスに必要な AWS IoT ポリシードキュメントを含むファイルを作成します。

      例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

      ```
      nano greengrass-v2-iot-role-alias-policy.json
      ```

      次の JSON をファイルにコピーします。

      - リソース ARN をロールエイリアスの ARN に置き換えます。

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:AssumeRoleWithCertificate",
      "Resource": "arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias"
    }
  ]
}
```

b. ポリシードキュメントから AWS IoT ポリシーを作成します。

- を作成するAWS IoTポリシーの名
  前*GreengrassCoreTokenExchangeRoleAliasPolicy*に置き換えます。

```
aws iot create-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy
 --policy-document file://greengrass-v2-iot-role-alias-policy.json
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "policyName": "GreengrassCoreTokenExchangeRoleAliasPolicy",
  "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassCoreTokenExchangeRoleAliasPolicy",
  "policyDocument": "{
    \\"Version\\":\\"2012-10-17\\",
    \\"Statement\\": [
      {
        \\"Effect\\": \\"Allow\\",
        \\"Action\\": \\"iot:AssumeRoleWithCertificate\\",
        \\"Resource\\": \\"arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias\\"
      }
    ]
  }",
  "policyVersionId": "1"
}
```

c. AWS IoT ポリシーを AWS IoT モノの証明書にアタッチします。

- をロールエイリアスAWS IoTポリシーの名前*GreengrassCoreTokenExchangeRoleAliasPolicy*に置き換えます。
- ターゲット ARN を AWS IoT モノの証明書の ARN に置き換えます。

```
aws iot attach-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy
 --target arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
```

要求が正常に処理された場合、コマンドは出力されません。

デバイスに証明書をダウンロードする

以前に、デバイスの証明書を開発用コンピュータにダウンロードしました。このセクションでは、Amazon ルート認証局 (CA) の証明書もダウンロードします。次に、開発用コンピュータとは別のコンピュータで Docker の AWS IoT Greengrass Core ソフトウェアを実行する場合は、証明書をそのホストコンピュータにコピーします。AWS IoT Greengrass Core ソフトウェアは、これらの証明書を使用して、AWS IoT クラウドサービスに接続します。

証明書をデバイスにダウンロードするには

1. 開発用コンピュータで、Amazon のルート認証局 (CA) 証明書をダウンロードします。デフォルトでは、AWS IoT 証明書が Amazon のルート CA 証明書と関連付けられています。

   Linux or Unix

   ```
   sudo curl -o ./greengrass-v2-certs/AmazonRootCA1.pem https://
   www.amazontrust.com/repository/AmazonRootCA1.pem
   ```

   Windows Command Prompt (CMD)

   ```
   curl -o .\greengrass-v2-certs\AmazonRootCA1.pem https://www.amazontrust.com/
   repository/AmazonRootCA1.pem
   ```

PowerShell

```
iwr -Uri https://www.amazontrust.com/repository/AmazonRootCA1.pem -OutFile .
\greengrass-v2-certs\AmazonRootCA1.pem
```

2. 開発用コンピュータとは別のデバイスで Docker の AWS IoT Greengrass Core ソフトウェアを実行する場合は、証明書をホストコンピュータにコピーします。開発用コンピュータとホストコンピュータで SSH と SCP が有効になっている場合、開発用コンピュータの scp コマンドを実行して、証明書を転送できます。をホストコンピュータの IP アドレス*device-ip-address*に置き換えます。

```
scp -r greengrass-v2-certs/ device-ip-address:~
```

設定ファイルを作成する

1. ホストコンピュータで、設定ファイルを配置するフォルダを作成します。

```
mkdir ./greengrass-v2-config
```

2. テキストエディタを使用して、./greengrass-v2-config フォルダに config.yaml という名前の設定ファイルを作成します。

   例えば、次のコマンドを実行し、GNU nano を使用して config.yaml を作成できます。

```
nano ./greengrass-v2-config/config.yaml
```

3. 次の YAML コンテンツをファイルにコピーします。この部分設定ファイルは、システムパラメータと Greengrass nucleus パラメータを指定します。

```
---
system:
  certificateFilePath: "/tmp/certs/device.pem.crt"
  privateKeyPath: "/tmp/certs/private.pem.key"
  rootCaPath: "/tmp/certs/AmazonRootCA1.pem"
  rootpath: "/greengrass/v2"
  thingName: "MyGreengrassCore"
services:
  aws.greengrass.Nucleus:
    componentType: "NUCLEUS"
```

```
    version: "nucleus-version"
    configuration:
      awsRegion: "region"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      iotDataEndpoint: "device-data-prefix-ats.iot.region.amazonaws.com"
      iotCredEndpoint: "device-credentials-prefix.credentials.region.amazonaws.com"
```

次に、以下の値を置き換えます。

- */tmp/certs*。コンテナの起動時にダウンロードした証明書をマウントする Docker コンテナ内のディレクトリ。
- */greengrass/v2*。 インストールに使用する Greengrass ルートフォルダ。GGC_ROOT 環境変数を使用して、この値を設定します。
- *MyGreengrassCore*。AWS IoT モノの名前。
- *nucleus #####*。インストールする AWS IoT Greengrass Core ソフトウェアのバージョン。この値は、ダウンロードした Docker イメージまたは Dockerfile のバージョンと一致する必要があります。latest タグ付きの Greengrass Docker イメージをダウンロードした場合は、**docker inspect *image-id*** を使用してイメージのバージョンを確認してください。
- *#####*。AWS IoT リソースを作成した AWS リージョン 。また、環境ファイルの AWS_REGION 環境変数に同じ値を指定する必要があります。
- *GreengrassCoreTokenExchangeRoleAlias*。トークン交換ロールエイリアス。
- *device-data-prefix*。AWS IoT データエンドポイントのプレフィックス。
- *device-credentials-prefix*。AWS IoT 認証情報エンドポイントのプレフィックス。

環境ファイルを作成する

このチュートリアルでは、環境ファイルを使用して、Docker コンテナ内の AWS IoT Greengrass Core ソフトウェアインストーラに渡される環境変数を設定します。また、docker run コマンドで -e または --env 引数を使用して、Docker コンテナに環境変数を設定する、または docker-compose.yml ファイルの environment ブロックで変数を設定することもできます。

1. テキストエディタを使用して、.env という名前の環境ファイルを作成します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用して現在のディレクトリに .env を作成できます。

```
nano .env
```

2. 次の内容をファイルにコピーします。

```
GGC_ROOT_PATH=/greengrass/v2
AWS_REGION=region
PROVISION=false
COMPONENT_DEFAULT_USER=ggc_user:ggc_group
INIT_CONFIG=/tmp/config/config.yaml
```

次に、以下の値を置き換えます。

- */greengrass/v2*。 AWS IoT Greengrass Core ソフトウェアのインストールに使用するルートフォルダへのパス。
- *#####*。AWS IoT リソースを作成した AWS リージョン 。[configuration file] (設定ファイル) の awsRegion 設定パラメータに同じ値を指定する必要があります。
- */tmp/config/*。Docker コンテナの起動時に設定ファイルをマウントするフォルダ。

> **Note**
>
> DEPLOY_DEV_TOOLS 環境変数を true に設定して、[Greengrass CLI component] (Greengrass CLI コンポーネント) をデプロイできます。これにより、Docker コンテナ内でカスタムコンポーネントを開発できます。このコンポーネントは、本番環境ではなく、開発環境でのみで使用することをお勧めします。このコンポーネントは、通常、本番環境では必要とされない情報や操作へのアクセスを提供します。このコンポーネントを必要なコアデバイスにのみデプロイして、最小特権の原則に従います。

コンテナで AWS IoT Greengrass Core ソフトウェアを実行する

このチュートリアルでは、Docker コンテナでビルドした Docker イメージを起動する方法を説明します。Docker CLI または Docker Compose CLI を使用して、Docker コンテナ内の AWS IoT Greengrass Core ソフトウェアイメージを実行できます。

Docker

- このチュートリアルでは、Docker コンテナでビルドした Docker イメージを起動する方法を説明します。

```
docker run --rm --init -it --name docker-image \
 -v path/to/greengrass-v2-config:/tmp/config/:ro \
 -v path/to/greengrass-v2-certs:/tmp/certs:ro \
 --env-file .env \
 -p 8883 \
 your-container-image:version
```

このコマンド例は、docker run に次の引数を使用します。

- --rm: コンテナの終了時にクリーンアップを実行します。
- --init: コンテナで init プロセスを使用します。

> **Note**
>
> Docker コンテナを停止するときに AWS IoT Greengrass Core ソフトウェアをシャットダウンするには、--init 引数が必要です。

- -it: (オプション) Docker コンテナを対話型プロセスとしてフォアグラウンドで実行します。これを -d 引数に置き換えて、代わりに Docker コンテナをデタッチモードで実行できます。詳細については、「Docker ドキュメント」の「デタッチ vs フォアグラウンド」を参照してください。
- --name: aws-iot-greengrass という名前のコンテナを実行します。
- -v: ボリュームを Docker コンテナにマウントして、コンテナ内で実行されている AWS IoT Greengrass で設定ファイルと証明書ファイルを利用可能にします。
- --env-file: (オプション) Docker コンテナ内の AWS IoT Greengrass Core ソフトウェアインストーラに渡される環境変数を設定する環境ファイルを指定します。この引数は、環境変数を設定するための [environment file] (環境ファイル) を作成した場合にのみ必要です。環境ファイルを作成していない場合は、--env 引数を使用して、「Docker 実行コマンド」で環境変数を直接設定できます。
- -p: (オプション) 8883 コンテナポートをホストマシンに公開します。AWS IoT Greengrass は MQTT トラフィックにポート 8883 を使用するため、MQTT を介して接続

および通信する場合は、この引数が必要です。他のポートを開くには、追加の `-p` 引数を使用します。

> **Note**
>
> セキュリティを強化して Docker コンテナを実行するには、`--cap-drop` 引数および `--cap-add` 引数を使用して、コンテナの Linux 機能を選択的に有効にします。詳細については、「Docker ドキュメント」の「[Runtime privilege and Linux capabilities] (ランタイム特権と Linux 機能)」を参照してください。

Docker Compose

1. テキストエディタを使用して、`docker-compose.yml` という名前の Docker Compose ファイルを作成します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用して現在のディレクトリに `docker-compose.yml` を作成できます。

   ```
   nano docker-compose.yml
   ```

   > **Note**
   >
   > から AWSが提供する Compose ファイルの最新バージョンをダウンロードして使用することもできますGitHub。

2. Compose ファイルに以下の内容を追加します。ファイルは次の例のようになります。*`your-container-name:version`* を Docker イメージの名前に置き換えます。

   ```
   version: '3.7'

   services:
     greengrass:
       init: true
       build:
         context: .
       container_name: aws-iot-greengrass
       image: your-container-name:version
       volumes:
   ```

```
      - /path/to/greengrass-v2-config:/tmp/config/:ro
      - /path/to/greengrass-v2-certs:/tmp/certs:ro
    env_file: .env
    ports:
      - "8883:8883"
```

この例の Compose ファイルでは、以下のパラメータはオプションです。

- `ports` - 8883 コンテナポートをホストマシンに公開します。AWS IoT Greengrass は MQTT トラフィックにポート 8883 を使用するため、MQTT を介して接続および通信する場合は、このパラメータが必要です。
- `env_file` - Docker コンテナ内の AWS IoT Greengrass Core ソフトウェアインストーラに渡される環境変数を設定する環境ファイルを指定します。このパラメータは、環境変数を設定するための [environment file] (環境ファイル) を作成した場合にのみ必要です。環境ファイルを作成していない場合は、[environment] (環境) パラメータを使用して、Compose ファイルで環境変数を直接設定できます。

> **Note**
>
> セキュリティを強化して Docker コンテナを実行するには、Compose ファイルで `cap_drop` および `cap_add` を使用して、コンテナの Linux 機能を選択的に有効にします。詳細については、「Docker ドキュメント」の「[Runtime privilege and Linux capabilities] (ランタイム特権と Linux 機能)」を参照してください。

3. 次のコマンドを実行して、コンテナを起動します。

```
docker-compose -f docker-compose.yml up
```

次のステップ

現在、Docker コンテナで AWS IoT Greengrass Core ソフトウェアが実行されています。次のコマンドを実行して、現在実行中のコンテナのコンテナ ID を取得します。

```
docker ps
```

次に、次のコマンドを実行してコンテナにアクセスし、コンテナ内で実行されている AWS IoT Greengrass Core ソフトウェアを検索できます。

```
docker exec -it container-id /bin/bash
```

単純なコンポーネントの作成については、「チュートリアル: AWS IoT Greengrass V2 の開始方法」の「ステップ 4: デバイス上でコンポーネントを開発およびテストする」を参照してください。

> **Note**
>
> `docker exec` を使用して Docker コンテナ内でコマンドを実行すると、これらのコマンドは Docker ログに記録されません。Docker ログにコマンドを記録するには、Docker コンテナに対話型シェルをアタッチします。詳細については、「インタラクティブシェルを Docker コンテナにアタッチする」を参照してください。

AWS IoT Greengrass Core ログファイルは `greengrass.log` と呼ばれ、*/greengrass/v2*/logs にあります。コンポーネントログファイルも同じディレクトリにあります。Greengrass ログをホストの一時ディレクトリにコピーするには、次のコマンドを実行します。

```
docker cp container-id:/greengrass/v2/logs /tmp/logs
```

コンテナの終了後、または削除後もログを保持する場合は、Greengrassディレクトリ全体をマウントするのではなく、*/greengrass/v2*/logs ディレクトリのみをホストの一時ログディレクトリにバインドマウントすることをお勧めします。詳細については、「Docker コンテナの外部で Greengrass ログを永続化する」を参照してください。

実行中の AWS IoT Greengrass Docker コンテナを停止するには、`docker stop` または `docker-compose -f docker-compose.yml stop` を実行します。このアクションは、`SIGTERM` を Greengrass プロセスに送信し、コンテナで開始されたすべての関連プロセスをシャットダウンします。Docker コンテナは、プロセス PID 1 として `docker-init` の実行可能ファイルで初期化されます。これは、残っているゾンビプロセスを削除するのに役立ちます。詳細については、Docker ドキュメントの「Specify an init process」を参照してください。

Docker コンテナで AWS IoT Greengrass を実行する際の問題のトラブルシューティングについては、「Docker コンテナでの AWS IoT Greengrass のトラブルシューティング」を参照してください。

Docker コンテナでの AWS IoT Greengrass のトラブルシューティング

以下の情報は、Docker コンテナで AWS IoT Greengrass を実行する際に生じる問題をトラブルシューティングし、Docker コンテナ内の AWS IoT Greengrass に関する問題をデバッグするために役立ちます。

トピック

Docker コンテナを実行する際に生じる問題のトラブルシューティング

以下の情報は、Docker コンテナでの AWS IoT Greengrass の実行に関する問題のトラブルシューティングに役立ちます。

トピック

エラー: Cannot perform an interactive login from a non TTY device (TTY 以外のデバイスから対話型ログインを実行できません)

`aws ecr get-login-password` コマンドを実行すると、このエラーが発生することがあります。最新の AWS CLI バージョン 2 またはバージョン 1 がインストールされているか確認してください。AWS CLI バージョン 2 を使用することをお勧めします。詳細については、「AWS Command Line Interface ユーザーガイド」の「AWS CLI のインストール」を参照してください。

エラー: 不明なオプション: -no-include-email

`aws ecr get-login` コマンドを実行すると、このエラーが発生することがあります。最新の AWS CLI バージョンがインストールされていることを確認します (例えば、`pip install awscli --upgrade --user` を実行します)。詳細については、「AWS Command Line Interface ユーザーガイド」の「Microsoft Windows に AWS Command Line Interface をインストールする」を参照してください。

エラー: A firewall is blocking file Sharing between windows and the containers. (ファイアウォールが、ウィンドウとコンテナー間のファイル共有をブロックしています。)

Windows コンピュータで Docker を実行すると、このエラーまたは `Firewall Detected` メッセージが表示されることがあります。このエラーは、仮想プライベートネットワーク (VPN) にサインインしているときにも発生する場合があり、ネットワーク設定が原因で共有ドライブをマウントできないことがあります。このような場合は、VPN をオフにし、Docker コンテナを再実行します。

エラー: GetAuthorizationToken オペレーションを呼び出すときにエラー (AccessDeniedException) が発生しました: ユーザー: arn:aws:iam::***account-id*** :user/<user-name> は実行する権限がありません: ecr:GetAuthorizationToken on resource: *

このエラーは、Amazon ECR リポジトリにアクセスするための十分な権限がない状態で `aws ecr get-login-password` コマンドを実行したときに表示されることがあります。詳細については、「Amazon ECR ユーザーガイド」の「Amazon ECR リポジトリポリシーの例」および「1 つの Amazon ECR リポジトリにアクセスする」を参照してください。

エラー: You have reached your pull rate limit (プルレート制限に達しました)

Docker Hub は、匿名ユーザーと無料の Docker Hub ユーザーが行うことができるプルリクエストの数を制限します。匿名ユーザーまたは無料のユーザーのプルリクエストの上限に達すると、次のいずれかのエラーが表示されます。

```
ERROR: toomanyrequests: Too Many Requests.
```

```
You have reached your pull rate limit.
```

これらのエラーを解決するには、別のプルリクエストを試行する前に数時間を置いてください。多数のプルリクエストを継続的に送信する予定がある場合は、Docker Hub ウェブサイトにある制限数に

関する情報と、Docker アカウントの認証とアップグレードのオプションに関する情報を参照してください。

Docker コンテナでの AWS IoT Greengrass のデバッグ

Docker コンテナの問題をデバッグするには、Greengrass ランタイムログを維持するか、Docker コンテナにインタラクティブシェルをアタッチすることができます。

Docker コンテナの外部で Greengrass ログを永続化する

AWS IoT Greengrass コンテナを停止した後、次の `docker cp` コマンドを使用して、Greengrass ログを Docker コンテナから一時ログディレクトリにコピーすることができます。

```
docker cp container-id:/greengrass/v2/logs /tmp/logs
```

コンテナが終了または削除された後にもログを永続させるには、*/greengrass/v2*/logs ディレクトリをバインドマウントした後に、AWS IoT Greengrass Docker コンテナを実行する必要があります。

/greengrass/v2/logs ディレクトリをバインドマウントするには、新しい AWS IoT Greengrass Docker コンテナを実行するときに次のいずれかを実行します。

- `docker run` コマンドに `-v` */tmp/logs*:*/greengrass/v2*/logs:ro を含めます。

  設定ファイル内の `volumes` ブロックを編集して、`docker-compose up` コマンドを実行する前に次の行を含めます。

  ```
  volumes:
   - /tmp/logs:/greengrass/v2/logs:ro
  ```

これで、Docker コンテナ内で AWS IoT Greengrass が実行されている間、ホストの */tmp/logs* でログを確認することができます。

Greengrass Docker コンテナを実行するための情報については、「手動プロビジョニングを使用して Docker で AWS IoT Greengrass を実行する」および「自動プロビジョニングを使用して Docker で AWS IoT Greengrass を実行する」を参照してください。

インタラクティブシェルを Docker コンテナにアタッチする

Docker コンテナ内でコマンドを実行するにあたり `docker exec` を使用する場合、これらのコマンドは Docker ログにキャプチャされません。コマンドを Docker ログに記録すると、Greengrass Docker コンテナの状態を調査する際に役立ちます。次のいずれかを行います:

- 別のターミナルで次のコマンドを実行して、ターミナルの標準入力、出力、およびエラーを実行中のコンテナにアタッチします。これにより、現在のターミナルから Docker コンテナを表示して、制御することができます。

  ```
  docker attach container-id
  ```

- 別のターミナルで次のコマンドを実行します。これにより、コンテナがアタッチされていない場合でも、コマンドをインタラクティブモードで実行できるようになります。

  ```
  docker exec -it container-id sh -c "command > /proc/1/fd/1"
  ```

一般的な AWS IoT Greengrass のトラブルシューティングヘルプについては、「トラブルシューティング」を参照してください。

AWS IoT Greengrass Core ソフトウェアを設定する

AWS IoT Greengrass Core ソフトウェアには、ソフトウェアの設定に使用できるオプションが用意されています。デプロイを作成して、各コアデバイスに AWS IoT Greengrass Core ソフトウェアを設定できます。

トピック

- Greengrass nucleus コンポーネントをデプロイする
- Greengrass nucleus をシステムサービスとして設定する
- JVM オプションでメモリ割り当てを制御する
- コンポーネントを実行するユーザーを設定する
- コンポーネントのシステムリソース制限を設定する
- ポート 443 での接続またはネットワークプロキシを通じた接続
- プライベート CA によって署名されたデバイス証明書を使用する
- MQTT タイムアウトとキャッシュ設定を設定する

Greengrass nucleus コンポーネントをデプロイする

AWS IoT Greengrass は、Greengrass AWS IoT Greengrass コアデバイスにデプロイできるコンポーネントとして Core ソフトウェアを提供します。複数の Greengrass コアデバイスに同じ設定を適用するためのデプロイを作成することができます。詳細については、Greengrass nucleusおよびAWS IoT Greengrass Core ソフトウェア (OTA) の更新を参照してください。

Greengrass nucleus をシステムサービスとして設定する

Core AWS IoT Greengrass ソフトウェアをデバイスの init システムのシステムサービスとして設定し、以下を実行する必要があります。

- デバイスの起動時に AWS IoT Greengrass Core ソフトウェアを起動します。これは、大量のデバイスのフリートを管理する場合に有効な方法です。
- プラグインコンポーネントをインストールして実行します。 AWSが提供するコンポーネントにはプラグインコンポーネントがいくつかあり、Greengrass nucleus と直接やり取りできます。コンポーネントタイプの詳細については、「コンポーネントタイプ」を参照してください。
- コアデバイスの AWS IoT Greengrass Core ソフトウェアに over-the-air (OTA) 更新を適用します。詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。
- デプロイでコンポーネントを新しいバージョンに更新したり、特定の設定パラメータを更新したりするときに、コンポーネントが AWS IoT Greengrass Core ソフトウェアまたはコアデバイスを再起動できるようにします。詳細については、「ブートストラップのライフサイクルステップ」を参照してください。

> **Important**
>
> Windows コアデバイスでは、 AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります。

トピック

- nucleus をシステムサービスとして設定する (Linux)
- nucleus をシステムサービスとして設定する (Windows)

nucleus をシステムサービスとして設定する (Linux)

Linux デバイスは、initd、systemd、SystemV などの様々な初期化システムをサポートします。AWS IoT Greengrass Core ソフトウェアをインストールするときに `--setup-system-service true`引数を使用し、nucleus をシステムサービスとして起動し、デバイスの起動時に起動するように設定します。インストーラは、 AWS IoT Greengrass Systemd で Core ソフトウェアをシステムサービスとして設定します。

nucleus がシステムサービスとして実行するように、手動で設定することもできます。次の例は、systemd のサービスファイルです。

```
[Unit]
Description=Greengrass Core

[Service]
Type=simple
PIDFile=/greengrass/v2/alts/loader.pid
RemainAfterExit=no
Restart=on-failure
RestartSec=10
ExecStart=/bin/sh /greengrass/v2/alts/current/distro/bin/loader

[Install]
WantedBy=multi-user.target
```

システムサービスを設定したら、次のコマンドを実行して、起動時のデバイスの起動と AWS IoT Greengrass Core ソフトウェアを起動または停止するように設定できます。

- サービス (systemd) のステータスを確認するには

  ```
  sudo systemctl status greengrass.service
  ```

- デバイスの起動時に nucleus を起動できるようにするには。

  ```
  sudo systemctl enable greengrass.service
  ```

- デバイスの起動時に nucleus の起動を停止するには。

  ```
  sudo systemctl disable greengrass.service
  ```

- AWS IoT Greengrass Core ソフトウェアを起動するには。

```
sudo systemctl start greengrass.service
```

- AWS IoT Greengrass Core ソフトウェアを停止するには。

```
sudo systemctl stop greengrass.service
```

nucleus をシステムサービスとして設定する (Windows)

AWS IoT Greengrass Core ソフトウェアをインストールするときに `--setup-system-service true`引数を使用し、nucleus を Windows サービスとして起動し、デバイスの起動時に起動するように設定します。

サービスを設定したら、次のコマンドを実行して、起動時のデバイスの起動と AWS IoT Greengrass Core ソフトウェアを起動または停止するように を設定できます。これらのコマンドを実行するには、コマンドプロンプトまたは管理者 PowerShell として を実行する必要があります。

Windows Command Prompt (CMD)

- サービスのステータスを確認するには

```
sc query "greengrass"
```

- デバイスの起動時に nucleus を起動できるようにするには。

```
sc config "greengrass" start=auto
```

- デバイスの起動時に nucleus の起動を停止するには。

```
sc config "greengrass" start=disabled
```

- AWS IoT Greengrass Core ソフトウェアを起動するには。

```
sc start "greengrass"
```

- AWS IoT Greengrass Core ソフトウェアを停止するには。

```
sc stop "greengrass"
```

> **Note**
>
> Windows デバイスでは、 AWS IoT Greengrass Core ソフトウェアは Greengrass コンポーネントプロセスをシャットダウンしている間、このシャットダウン信号を無視します。このコマンドの実行時に AWS IoT Greengrass Core ソフトウェアがシャットダウン信号を無視する場合は、数秒待ってから再試行してください。

PowerShell

- サービスのステータスを確認するには

```
Get-Service -Name "greengrass"
```

- デバイスの起動時に nucleus を起動できるようにするには。

```
Set-Service -Name "greengrass" -Status stopped -StartupType automatic
```

- デバイスの起動時に nucleus の起動を停止するには。

```
Set-Service -Name "greengrass" -Status stopped -StartupType disabled
```

- AWS IoT Greengrass Core ソフトウェアを起動するには。

```
Start-Service -Name "greengrass"
```

- AWS IoT Greengrass Core ソフトウェアを停止するには。

```
Stop-Service -Name "greengrass"
```

> **Note**
>
> Windows デバイスでは、 AWS IoT Greengrass Core ソフトウェアは Greengrass コンポーネントプロセスをシャットダウンしている間、このシャットダウン信号を無視します。このコマンドの実行時に AWS IoT Greengrass Core ソフトウェアがシャットダウン信号を無視する場合は、数秒待ってから再試行してください。

JVM オプションでメモリ割り当てを制御する

メモリが制限されている AWS IoT Greengrass デバイスで実行している場合は、Java 仮想マシン (JVM) オプションを使用して、最大ヒープサイズ、ガベージコレクションモード、コンパイラオプションを制御できます。これにより、 AWS IoT Greengrass Core ソフトウェアが使用するメモリ量が制御されます。JVM のヒープサイズは、ガベージコレクションが発生する前、あるいはアプリケーションがメモリ不足になる前に、アプリケーションが使用できるメモリ量を決定します。最大ヒープサイズは、多量の作業を実行中にヒープを拡張する場合に JVM が割り当てることのできる最大メモリ容量を指定します。

メモリの割り当てを制御するには、新しいデプロイを作成するか、nucleus コンポーネントが含まれる既存のデプロイを修正し、nucleus コンポーネント設定内の `jvmOptions` 設定パラメータで JVM オプションを指定します。

要件に応じて、メモリ割り当てを減らしたり、最小限のメモリ割り当てで AWS IoT Greengrass Core ソフトウェアを実行できます。

メモリ割り当ての削減

メモリ割り当てを減らして AWS IoT Greengrass Core ソフトウェアを実行するには、次の設定マージ更新の例を使用して nucleus 設定で JVM オプションを設定することをお勧めします。

```
{
  "jvmOptions": "-Xmx64m -XX:+UseSerialGC -XX:TieredStopAtLevel=1"
}
```

最小限のメモリ割り当て

最小メモリ割り当てで AWS IoT Greengrass Core ソフトウェアを実行するには、次の設定マージ更新の例を使用して nucleus 設定で JVM オプションを設定することをお勧めします。

```
{
  "jvmOptions": "-Xmx32m -XX:+UseSerialGC -Xint"
}
```

これらの設定マージ更新例では、次の JVM オプションを使用します。

`-Xmx`*NN*`m`

JVM の最大ヒープサイズを設定します。

メモリの割り当てを減らす場合は、`-Xmx64m` を開始値として指定し、ヒープサイズを 64 MB に制限します。最小限のメモリ割り当てにする場合は、`-Xmx32m` を開始値として指定し、ヒープサイズを 32 MB に制限します。

実際の要件に応じて、`-Xmx` の値は増減することができますが、最大ヒープサイズは 16 MB 以下に設定しないことを強く推奨します。最大ヒープサイズが環境に対して低すぎる場合、メモリ不足により AWS IoT Greengrass Core ソフトウェアで予期しないエラーが発生する可能性があります。

`-XX:+UseSerialGC`

JVM ヒープスペースにシリアルガベージコレクションを使用するように指定します。シリアルガベージコレクタは速度が遅くなりますが、他の JVM ガベージコレクションの実装と比べると使用するメモリが少なくなります。

`-XX:TieredStopAtLevel=1`

Java just-in-time (JIT) コンパイラを 1 回使用するように JVM に指示します。JIT でコンパイルされたコードはデバイスメモリ内のスペースを使用するため、JIT コンパイラを複数回使用すると、1 回のコンパイルよりも多くのメモリを消費します。

`-Xint`

just-in-time (JIT) コンパイラを使用しないように JVM に指示します。代わりに、JVM は解釈専用モードで実行されます。このモードは JIT コンパイルコードを実行する場合よりも速度は遅くなりますが、コンパイルされたコードはメモリ内のスペースを使用しません。

設定マージ更新の作成については、「コンポーネント設定の更新」を参照してください。

コンポーネントを実行するユーザーを設定する

AWS IoT Greengrass Core ソフトウェアは、ソフトウェアを実行するものとは異なるシステムユーザーおよびグループとしてコンポーネントプロセスを実行できます。これにより、 AWS IoT Greengrass コアデバイスで実行されるコンポーネントにアクセス許可を付与することなく、 Core ソフトウェアをルートまたは管理者ユーザーとして実行できるため、セキュリティが向上します。

次の表は、指定したユーザーとして AWS IoT Greengrass Core ソフトウェアが実行できるコンポーネントのタイプを示しています。詳細については、「コンポーネントタイプ」を参照してください。

コンポーネントタイプ	コンポーネントユーザを設定する
nucleus	 いいえ
プラグイン	 いいえ
ジェネリック	 はい
Lambda (非コンテナ化)	 はい
Lambda (コンテナ化)	 はい

デプロイ設定で指定する前に、コンポーネントユーザーを作成する必要があります。Windows ベースのデバイスでは、 LocalSystem アカウントの認証情報マネージャーインスタンスにユーザーのユーザー名とパスワードを保存する必要があります。詳細については、「Windows デバイスでコンポーネントユーザーをセットアップする」を参照してください。

Linux ベースのデバイスでコンポーネントユーザーを設定する場合、オプションでグループを指定することもできます。ユーザーとグループを *user*:*group* の形式に従ってコロン (:) で区切って指定します。グループを指定しない場合、 AWS IoT Greengrass Core ソフトウェアはデフォルトでユー

ザーのプライマリグループになります。名前または ID を使用して、ユーザーとグループを識別できます。

Linux ベースのデバイスでは、コンポーネントを存在しないシステムユーザー (不明なユーザーとも呼ばれる) として実行することで、セキュリティを強化することもできます。Linux プロセスは、同じユーザーによって実行される他のプロセスに信号を送ることができます。不明なユーザーは他のプロセスを実行しないため、コンポーネントを不明なユーザーとして実行して、コンポーネントがコアデバイス上の他のコンポーネントに信号を送るのを防ぐことができます。コンポーネントを不明なユーザーとして実行するには、コアデバイスに存在しないユーザー ID を指定します。不明なグループとして実行する場合は、存在しないグループ ID を指定することもできます。

各コンポーネントと各コアデバイスごとに、ユーザーを設定することができます。

- コンポーネントの設定

  各コンポーネントが、そのコンポーネントに固有のユーザーで実行されるように設定することができます。デプロイを作成するときに、各コンポーネントのユーザをそのコンポーネントの `runWith` 設定で指定できます。 AWS IoT Greengrass Core ソフトウェアは、コンポーネントを設定すると、指定されたユーザーとしてコンポーネントを実行します。設定がない場合は、コアデバイス用に設定したデフォルトのユーザーとしてコンポーネントを実行します。デプロイ設定でコンポーネントユーザーを指定するための詳細については、デプロイの作成 の `runWith` 設定パラメータを参照してください。

- コアデバイスのデフォルトユーザーを設定する

  AWS IoT Greengrass Core ソフトウェアがコンポーネントの実行に使用するデフォルトユーザーを設定できます。 AWS IoT Greengrass Core ソフトウェアは、コンポーネントを実行するときに、そのコンポーネントのユーザーを指定しているかどうかを確認し、それを使用してコンポーネントを実行します。コンポーネントがユーザーを指定しない場合、 AWS IoT Greengrass Core ソフトウェアはコアデバイスに設定したデフォルトユーザーとしてコンポーネントを実行します。詳細については、「デフォルトのコンポーネントユーザーを設定する」を参照してください。

> **Note**
>
> Windows ベースのデバイスでは、コンポーネントを実行するにあたり、少なくとも一人のデフォルトユーザーを指定する必要があります。
>
> Linux ベースのデバイスでは、コンポーネントを実行するユーザーを設定しない場合には、以下について考慮する必要があります。

- AWS IoT Greengrass Core ソフトウェアをルートとして実行すると、ソフトウェアはコンポーネントを実行しません。root として実行する場合は、コンポーネントを実行するデフォルトユーザーを指定する必要があります。
- AWS IoT Greengrass Core ソフトウェアを非ルートユーザーとして実行する場合、ソフトウェアはそのユーザーとしてコンポーネントを実行します。

トピック

Windows デバイスでコンポーネントユーザーをセットアップする

Windows デバイスでコンポーネントユーザーをセットアップするには

1. デバイスの LocalSystem アカウントにコンポーネントユーザーを作成します。

```
net user /add component-user password
```

2. Microsoft の PsExec ユーティリティを使用して、コンポーネントユーザーのユーザー名とパスワードを LocalSystem アカウントの認証情報マネージャーインスタンスに保存します。

```
psexec -s cmd /c cmdkey /generic:component-user /user:component-user /pass:password
```

> **Note**
>
> Windows ベースのデバイスでは、 LocalSystem アカウントは Greengrass nucleus を実行します。コンポーネントユーザー情報を LocalSystemアカウントに保存するには、PsExec ユーティリティを使用する必要があります。認証情報マネージャーアプリケーションを使用すると、この情報は、 アカウントではなく、現在ログオンしているユーザーの Windows LocalSystemアカウントに保存されます。

デフォルトのコンポーネントユーザーを設定する

デプロイを使用して、コアデバイスのデフォルトユーザーを設定できます。このデプロイでは、nucleus コンポーネントの設定を更新します。

> **Note**
>
> `--component-default-user` オプションを使用して AWS IoT Greengrass Core ソフトウェアをインストールするときに、デフォルトユーザーを設定することもできます。詳細については、「AWS IoT Greengrass Core ソフトウェアをインストールします。」を参照してください。

`aws.greengrass.Nucleus` コンポーネントに対する、以下の設定更新を指定するデプロイを作成します。

Linux

```
{
  "runWithDefault": {
    "posixUser": "ggc_user:ggc_group"
  }
}
```

Windows

```
{
  "runWithDefault": {
    "windowsUser": "ggc_user"
  }
}
```

> **Note**
>
> 指定するユーザーは存在している必要があり、このユーザーのユーザー名とパスワードは Windows デバイスの LocalSystem アカウントの認証情報マネージャーインスタンスに保存されている必要があります。詳細については、「Windows デバイスでコンポーネントユーザーをセットアップする」を参照してください。

次の例では、ggc_user をデフォルトユーザー、および ggc_group をデフォルトグループとして設定する Linux ベースのデバイスのデプロイを定義しています。merge 設定の更新には、シリアル化された JSON オブジェクトが必要です。

```
{
  "components": {
    "aws.greengrass.Nucleus": {
      "version": "2.12.2",
      "configurationUpdate": {
        "merge": "{\"runWithDefault\":{\"posixUser\":\"ggc_user:ggc_group\"}}"
      }
    }
  }
}
```

コンポーネントのシステムリソース制限を設定する

> **Note**
>
> この機能は、Greengrass nucleus コンポーネントの v2.4.0 以降で使用できます。 AWS IoT Greengrass 現在、Windows コアデバイスでこの機能をサポートしていません。

各コンポーネントのプロセスがコアデバイスで使用できる CPU および RAM の最大使用数を設定できます。

下表は、システムリソースの制限をサポートするコンポーネントの種類を示したものです。詳細については、「コンポーネントタイプ」を参照してください。

コンポーネントタイプ	システムリソースの制限を設定する
nucleus	いいえ

コンポーネントタイプ	システムリソースの制限を設定する
プラグイン	 いいえ
ジェネリック	 はい
Lambda (非コンテナ化)	 はい
Lambda (コンテナ化)	 いいえ

> **Important**
>
> Docker コンテナで AWS IoT Greengrass Core ソフトウェアを実行する場合、システムリソースの制限はサポートされません。

各コンポーネントと各コアデバイスごとに、システムリソースの制限を設定することができます。

- コンポーネントの設定

  各コンポーネントは、そのコンポーネントに固有のシステムリソース制限を設定することができます。デプロイを作成するときに、デプロイ内の各コンポーネントに対して、システムリソースの制限を指定することができます。コンポーネントがシステムリソース制限をサポートしている場合、 AWS IoT Greengrass Core ソフトウェアは、コンポーネントのプロセスにその制限を適用します。コンポーネントのシステムリソース制限を指定しない場合、 AWS IoT Greengrass Core ソ

フトウェアはコアデバイスに設定したデフォルトを使用します。詳細については、「デプロイの作成」を参照してください。

- コアデバイスのデフォルトを設定する

  AWS IoT Greengrass Core ソフトウェアがこれらの制限をサポートするコンポーネントに適用するデフォルトのシステムリソース制限を設定できます。 AWS IoT Greengrass Core ソフトウェアがコンポーネントを実行すると、そのコンポーネントに指定したシステムリソース制限が適用されます。そのコンポーネントがシステムリソース制限を指定しない場合、 AWS IoT Greengrass Core ソフトウェアは、コアデバイス用に設定したデフォルトのシステムリソース制限を適用します。デフォルトのシステムリソース制限を指定しない場合、 AWS IoT Greengrass Core ソフトウェアはデフォルトでシステムリソース制限を適用しません。詳細については、「デフォルトのシステムリソース制限を設定する」を参照してください。

デフォルトのシステムリソース制限を設定する

Greengrass nucleus コンポーネントをデプロイして、コアデバイスのデフォルトのシステムリソース制限を設定することができます。デフォルトのシステムリソース制限を設定するには、`aws.greengrass.Nucleus` コンポーネントに対して次の設定更新を指定するデプロイを作成します。

```
{
  "runWithDefault": {
    "systemResourceLimits": {
      "cpu": cpuTimeLimit,
      "memory": memoryLimitInKb
    }
  }
}
```

次の例では、CPU 時間制限を 2 (4 つの CPU コアを持つデバイスでの使用量 50% に相当) に設定するデプロイを定義しています。また、この例ではメモリ使用量を 100 MB に設定しています。

```
{
  "components": {
    "aws.greengrass.Nucleus": {
      "version": "2.12.2",
      "configurationUpdate": {
        "merge": "{\"runWithDefault\":{\"systemResourceLimits\":\"cpus\":2,\"memory
\":102400}}}"
```

```
        }
      }
    }
}
```

ポート 443 での接続またはネットワークプロキシを通じた接続

AWS IoT Greengrass コアデバイスは、TLS クライアント認証で MQTT メッセージングプロトコル AWS IoT Core を使用して と通信します。慣例では、TLS を介した MQTT ではポート 8883 を使用します。ただし、セキュリティ対策として、制限の厳しい環境では一定範囲の TCP ポートに対するインバウンドトラフィックとアウトバウンドトラフィックを制限する場合があります。例えば、企業のファイアウォールでは HTTPS トラフィック用のポート 443 は開いても、あまり一般的ではないプロトコル用の他のポート (MQTT トラフィック用のポート 8883 など) は閉じる場合があります。他にも、制限のある環境によっては、インターネットに接続する前に、すべてのトラフィックがプロキシを経由する必要が生じる場合もあります。

> **Note**
>
> Greengrass nucleus コンポーネント v2.0.3 以前を実行する Greengrass コアデバイスは、ポート 8443 を使用して AWS IoT Greengrass データプレーンエンドポイントに接続します。これらのデバイスは、ポート 8443 でこのエンドポイントに接続できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

これらのシナリオで通信を有効にするために、 AWS IoT Greengrass には以下の設定オプションが用意されています。

- ポート 443 を介した MQTT 通信。ネットワークでポート 443 への接続を許可している場合は、デフォルトのポート 8883 ではなくポート 443 を MQTT トラフィックに使用するように、Greengrass コアデバイスを設定することができます。ポート 443 への直接接続またはネットワークプロキシサーバーを介した接続を使用できます。証明書ベースのクライアント認証を使用するデフォルト設定とは異なり、ポート 443 上の MQTT は認証にデバイスのサービスロールを使用します。

  詳細については、「ポート 443 経由で MQTT を設定する」を参照してください。
- ポート 443 を介した HTTPS 通信。 AWS IoT Greengrass Core ソフトウェアは、デフォルトでポート 8443 経由で HTTPS トラフィックを送信しますが、ポート 443 を使用するように設定でき

ます。 AWS IoT Greengrass は、Application Layer Protocol Network (ALPN) TLS 拡張を使用してこの接続を有効にします。デフォルト設定と同様に、ポート 443 経由の HTTPS は証明書ベースのクライアント認証を使用します。

> **Important**
>
> ALPN を使用したポート 443 経由の HTTPS 通信を有効にするためには、コアデバイスで Java 8 更新 252 以降が実行されている必要があります。Java バージョン 9 以降のすべての更新も、ALPN をサポートしています。

詳細については、「ポート 443 経由で HTTPS を設定する」を参照してください。

- ネットワークプロキシを介した接続。ネットワークプロキシサーバーを Greengrass コアデバイスに接続するための仲介役として設定できます。 AWS IoT Greengrass は、HTTP および HTTPS プロキシに対する基本認証をサポートしています。

  HTTPS プロキシを使用するには、Greengrass コアデバイスで Greengrass nucleus v2.5.0 以降を実行している必要があります。

  AWS IoT Greengrass Core ソフトウェアは、、、ALL_PROXY、HTTP_PROXY、HTTPS_PROXYおよび NO_PROXY環境変数を介してプロキシ設定をコンポーネントに渡します。コンポーネントはこれらの設定を使用して、プロキシ経由で接続する必要があります。コンポーネントは、一般的なライブラリ (boto3、cURL、python `requests` パッケージなど) を使用し、通常、これらの環境変数をデフォルトで使用して接続を行います。関数もこれらの同じ環境変数を指定した場合、 AWS IoT Greengrass ではオーバーライドされません。

  詳細については、「ネットワークプロキシを設定する」を参照してください。

ポート 443 経由で MQTT を設定する

既存のコアデバイスに、または新しいコアデバイスに AWS IoT Greengrass Core ソフトウェアをインストールする際に、ポート 443 経由で MQTT を設定することができます。

トピック

- 既存のコアデバイスにポート 443 経由で MQTT を設定する
- インストール中にポート 443 経由で MQTT を設定する

既存のコアデバイスにポート 443 経由で MQTT を設定する

デプロイを使用して、単一のコアデバイスまたはコアデバイスのグループに MQTT をポート 443 経由で設定できます。このデプロイでは、nucleus コンポーネントの設定を更新します。mqtt 設定を更新すると、nucleus が再起動します。

ポート 443経由で MQTT を設定するには、aws.greengrass.Nucleus コンポーネントに以下の設定更新を指定するデプロイを作成します。

```
{
  "mqtt": {
    "port": 443
  }
}
```

次の例では、ポート 443 経由で MQTT を設定するデプロイを定義します。merge 設定の更新には、シリアル化された JSON オブジェクトが必要です。

```
{
  "components": {
    "aws.greengrass.Nucleus": {
      "version": "2.12.2",
      "configurationUpdate": {
        "merge": "{\"mqtt\":{\"port\":443}}"
      }
    }
  }
}
```

インストール中にポート 443 経由で MQTT を設定する

コアデバイスに AWS IoT Greengrass Core ソフトウェアをインストールするときに、ポート 443 経由で MQTT を設定できます。ポート 443 経由で MQTT を設定するには、--init-config インストーラ引数を使用します。この引数は、手動によるプロビジョニング、フリートプロビジョニング、またはカスタムプロビジョニングでインストールするときに指定できます。

ポート 443 経由で HTTPS を設定する

この機能を使用するには、Greengrass nucleus v2.0.4 以降が必要です。

既存のコアデバイスに、または新しいコアデバイスに AWS IoT Greengrass Core ソフトウェアをインストールする際に、ポート 443 経由で HTTPS を設定することができます。

トピック

- 既存のコアデバイスにポート 443 経由で HTTPS を設定する
- インストール中にポート 443 経由で HTTPS を設定する

既存のコアデバイスにポート 443 経由で HTTPS を設定する

デプロイを使用して、単一のコアデバイスまたはコアデバイスのグループに HTTPS をポート 443 経由で設定できます。このデプロイでは、nucleus コンポーネントの設定を更新します。

ポート 443 経由で HTTPS を設定するには、`aws.greengrass.Nucleus` コンポーネントに以下の設定更新を指定するデプロイを作成します。

```
{
  "greengrassDataPlanePort": 443
}
```

次の例では、ポート 443 経由で HTTPS を設定するデプロイを定義します。`merge` 設定の更新には、シリアル化された JSON オブジェクトが必要です。

```
{
  "components": {
    "aws.greengrass.Nucleus": {
      "version": "2.12.2",
      "configurationUpdate": {
        "merge": "{\"greengrassDataPlanePort\":443}"
      }
    }
  }
}
```

インストール中にポート 443 経由で HTTPS を設定する

Core ソフトウェアを AWS IoT Greengrass コアデバイスにインストールするときに、ポート 443 経由で HTTPS を設定できます。ポート 443 経由で HTTPS を設定するには、`--init-config` インストーラ引数を使用します。この引数は、手動によるプロビジョニング、フリートプロビジョニング、またはカスタムプロビジョニングでインストールするときに指定できます。

ネットワークプロキシを設定する

このセクションの手順に従って、Greengrass コアデバイスが HTTP または HTTPS ネットワークプロキシを介してインターネットに接続するように設定します。コアデバイスが使用するエンドポイントとポートの詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

Important

コアデバイスで Greengrass nucleus v2.4.0 以前のバージョンが動作している場合、ネットワークプロキシを使用するには、デバイスのロールが以下の権限を許可している必要があります。

- `iot:Connect`
- `iot:Publish`
- `iot:Receive`
- `iot:Subscribe`

これは、デバイスがトークン交換サービスの AWS 認証情報を使用して への MQTT 接続を認証するため必要です AWS IoT。デバイスは MQTT を使用して からデプロイを受信してインストールするため AWS クラウド、ロールでこれらのアクセス許可を定義しない限り、デバイスは機能しません。デバイスは通常 MQTT 接続を認証するために X.509 証明書を使用しますが、プロキシを使用している場合には、認証にこれを使用できません。
デバイスのロールを設定する方法の詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

トピック

既存のコアデバイスでネットワークプロキシを設定する

デプロイを使用して、単一のコアデバイスまたはコアデバイスのグループにネットワークプロキシを設定できます。このデプロイでは、nucleus コンポーネントの設定を更新します。networkProxy 設定を更新すると、nucleus が再起動します。

ネットワークプロキシを設定するには、aws.greengrass.Nucleus コンポーネントに以下の設定更新をマージするデプロイを作成します。この設定更新プログラムには、networkProxy オブジェクトが含まれます。

```
{
  "networkProxy": {
    "noProxyAddresses": "http://192.168.0.1,www.example.com",
    "proxy": {
      "url": "https://my-proxy-server:1100"
    }
  }
}
```

次の例では、ネットワークプロキシを設定するデプロイを定義します。merge 設定の更新には、シリアル化された JSON オブジェクトが必要です。

```
{
  "components": {
    "aws.greengrass.Nucleus": {
      "version": "2.12.2",
      "configurationUpdate": {
        "merge": "{\"networkProxy\":{\"noProxyAddresses\":
\"http://192.168.0.1,www.example.com\",\"proxy\":{\"url\":\"https://my-proxy-
server:1100\",\"username\":\"Mary_Major\",\"password\":\"pass@word1357\"}}}"
      }
    }
  }
}
```

インストール中にネットワークプロキシを設定する

コアデバイスに AWS IoT Greengrass Core ソフトウェアをインストールするときに、ネットワークプロキシを設定できます。--init-config インストーラ引数を使用して、ネットワークプロキシを設定します。この引数は、手動によるプロビジョニング、フリートプロビジョニング、またはカスタムプロビジョニングでインストールするときに指定できます。

コアデバイスが HTTPS プロキシを信頼できるようにする

HTTPS プロキシを使用するようにコアデバイスを設定する場合は、コアデバイスにプロキシサーバー証明書チェーンを追加して、コアデバイスが HTTPS プロキシを信頼するようにする必要があります。この設定を行わなかった場合、コアデバイスがプロキシ経由でトラフィックをルーティングしようとしたときにエラーが発生する可能性があります。コアデバイスの Amazon ルート CA 証明書ファイルにプロキシサーバー CA 証明書を追加します。

コアデバイスに HTTPS プロキシを信頼させるには

1. コアデバイス上で Amazon ルート CA 証明書ファイルを検索します。

   - 自動プロビジョニングを使用して AWS IoT Greengrass Core ソフトウェアをインストールした場合、Amazon ルート CA 証明書ファイルは にあります*/greengrass/v2*/rootCA.pem。
   - 手動またはフリートプロビジョニングを使用して AWS IoT Greengrass Core ソフトウェアをインストールした場合、Amazon ルート CA 証明書ファイルは に存在する可能性があります*/greengrass/v2*/AmazonRootCA1.pem。

   Amazon ルート CA 証明書がこれらの場所に存在しない場合は、*/greengrass/v2*/config/effectiveConfig.yaml の system.rootCaPath プロパティを確認して場所を見つけます。

2. Amazon ルート CA 証明書ファイルにプロキシサーバー CA 証明書ファイルの内容を追加します。

   次の例は、Amazon ルート CA 証明書ファイルに追加されたプロキシサーバー CA 証明書を示しています。

```
-----BEGIN CERTIFICATE-----
MIIEFTCCAv2gAwIQWgIVAMHSAzWG/5YVRYtRQOxXUTEpHuEmApzGCSqGSIb3DQEK
\nCwUAhuL9MQswCQwJVUzEPMAVUzEYMBYGA1UECgwP1hem9uLmNvbSBJbmMuMRww
... content of proxy CA certificate ...
+vHIRlt0e5JAm5\noTIZGoFbK82A0/nO7f/t5PSIDAim9V3Gc3pSXxCCAQoFYnui
GaPUlGk1gCE84a0X\n7Rp/lND/PuMZ/s8YjlkY2NmYmNjMCAXDTE5MTEyN2cM216
gJMIADggEPADf2/m45hzEXAMPLE=
-----END CERTIFICATE-----

-----BEGIN CERTIFICATE-----
MIIDQTCCAimgF6AwIBAgITBmyfz/5mjAo54vB4ikPmljZKyjANJmApzyMZFo6qBg
```

```
ADA5MQswCQYDVQQGEwJVUzEPMA0tMVT8QtPHRh8jrdkGA1UEChMGDV3QQDExBBKW
... content of root CA certificate ...
o/ufQJQWUCyziar1hem9uMRkwFwYVPSHCb2XV4cdFyQzR1KldZwgJcIQ6XUDgHaa
5MsI+yMRQ+hDaXJiobldXgjUka642M4UwtBV8oK2xJNDd2ZhwLnoQdeXeGADKkpy
rqXRfKoQnoZsG4q5WTP46EXAMPLE
-----END CERTIFICATE-----
```

networkProxy オブジェクト

ネットワークプロキシに関する情報を指定するには、networkProxy オブジェクトを使用します。このオブジェクトには、次の情報が含まれます。

noProxyAddresses

(オプション) プロキシの対象外となる IP アドレスやホスト名をカンマで区切ったリスト。

proxy

接続先のプロキシ。このオブジェクトには、次の情報が含まれます。

url

プロキシサーバーの URL (scheme://userinfo@host:port 形式)。

- scheme - スキーム。http または https である必要があります。

  > **Important**
  >
  > HTTPS プロキシを使用するには、Greengrass コアデバイスで Greengrass nucleus v2.5.0 以降を実行している必要があります。
  > HTTPS プロキシを設定する場合は、コアデバイスの Amazon ルート CA 証明書にプロキシサーバー CA 証明書を追加する必要があります。詳細については、「コアデバイスが HTTPS プロキシを信頼できるようにする」を参照してください。

- userinfo - (オプション) ユーザー名とパスワードの情報。この情報を url で指定する場合、Greengrass コアデバイスは username および password フィールドを無視します。
- host – プロキシサーバーのホスト名または IP アドレス。
- port - (オプション) ポート番号。ポートを指定しない場合、Greengrass コアデバイスは次のデフォルト値を使用します。
  - http – 80
  - https – 443

`username`

(オプション) プロキシサーバーを認証するユーザー名です。

`password`

(オプション) プロキシサーバーを認証するパスワードです。

プライベート CA によって署名されたデバイス証明書を使用する

カスタムのプライベート認証機関 (CA) を使用する場合は、 Greengrass nucleus の **`greengrassDataPlaneEndpoint`** で **`iotdata`** を設定する必要があります。このオプションは、**`--init-config`** インストーラ引数を使用するデプロイまたはインストール中に設定できます。

デバイスが接続する Greengrass データプレーンのエンドポイントを、カスタマイズすることができます。この設定オプションを **`iotdata`** に設定して、Greengrass データプレーンエンドポイントを (**`iotDataEndpoint`** により指定することが可能な) IoT データエンドポイントと同じエンドポイントに設定できます。

MQTT タイムアウトとキャッシュ設定を設定する

AWS IoT Greengrass 環境では、コンポーネントは MQTT を使用して と通信できます AWS IoT Core。 AWS IoT Greengrass Core ソフトウェアは、コンポーネントの MQTT メッセージを管理します。コアデバイスが AWS クラウド への接続を失うと、ソフトウェアは MQTT メッセージをキャッシュして、接続が復元されたときに後で再試行します。メッセージのタイムアウトやキャッシュのサイズなどの設定を設定できます。詳細については、「Greengrass nucleus コンポーネント」の「`mqtt` および `mqtt.spooler` 設定パラメータ」を参照してください。

AWS IoT Core は、MQTT メッセージブローカーにサービスクォータを課します。これらのクォータは、コアデバイスと AWS IoT Core 間に送信されるメッセージに適用される場合があります。詳細については、「AWS 全般のリファレンス」の「AWS IoT Core メッセージブローカーのサービスクォータ」を参照ください。

AWS IoT Greengrass Core ソフトウェア (OTA) の更新

AWS IoT Greengrass Core ソフトウェアは、Greengrass nucleus コンポーネントと、ソフトウェアの更新 over-the-air (OTA) を実行するためにデバイスにデプロイできるその他のオプションコンポーネントで構成されます。この機能は AWS IoT Greengrass Core ソフトウェアに組み込まれています。

OTA 更新により、以下の作業の効率が向上します。

- セキュリティの脆弱性を修正する。
- ソフトウェアの安定性の問題に対処する。
- 新しい機能や改良された機能をデプロイする。

トピック

要件

AWS IoT Greengrass Core ソフトウェアに対する OTA 更新をデプロイするため、次の要件が適用されます:

- Greengrass コアデバイスはデプロイを受信するため、AWS クラウド への接続が必要です。
- Greengrass コアデバイスは、AWS IoT Core と AWS IoT Greengrass を使用した認証用の証明書とキーで正しく設定およびプロビジョニングする必要があります。
- AWS IoT Greengrass Core ソフトウェアは、システムサービスとしてセットアップして実行する必要があります。nucleus を JAR ファイル、`Greengrass.jar` から実行する場合、OTA 更新は機能しません。詳細については、「Greengrass nucleus をシステムサービスとして設定する」を参照してください。

コアデバイスの考慮事項

OTA 更新を実行する前に、更新するコアデバイスと接続されているクライアントデバイスへの影響に注意してください:

- Greengrass nucleus がシャットダウンします。
- コアデバイスで実行されているすべてのコンポーネントもシャットダウンされます。これらのコンポーネントがローカルリソースに書き込む場合、それらのリソースを適切にシャットダウンしない限り、正しくない状態で放置する場合があります。コンポーネントは、使用するリソースをクリー

ンアップするまで、プロセス間通信を使用して、nucleus コンポーネントに更新を延期するように指示します。

- nucleus コンポーネントがシャットダウン中、コアデバイスは AWS クラウド とローカルデバイスとの接続を失います。コアデバイスは、シャットダウン中にクライアントデバイスからのメッセージをルーティングしません。
- コンポーネントとして実行される存続期間の長い Lambda 関数は、動的状態の情報を失って保留中の作業はすべて破棄されます。

Greengrass nucleus の更新動作

コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。

Greengrass nucleus コンポーネントのバージョンが変更されるとき、AWS IoT Greengrass Core ソフトウェア (デバイスの nucleus と他のすべてのコンポーネントを含む) が再起動して変更を適用します。nucleus コンポーネントが更新される際のコアデバイスへの影響によって、新しい nucleus パッチバージョンがデバイスにデプロイされるタイミングを制御することをお勧めします。それを実行するには、Greengrass nucleus コンポーネントを直接デプロイに含める必要があります。コンポーネントを直接含めるということは、そのコンポーネントの特定バージョンをデプロイ設定に含め、コンポーネントの従属関係に依存してそのコンポーネントをデバイスにデプロイしないことを意味します。コンポーネントレシピで従属関係を定義する方法の詳細については、「レシピの形式」を参照してください。

次の表を参照して、アクションとデプロイ設定に基づいて Greengrass nucleus コンポーネントの更新動作について把握します。

[アクション]	デプロイ設定	nucleus の更新動作
デプロイを改訂せずに、既存のデプロイがターゲットするモノグループに新しいデバイスを追加します。	デプロイは Greengrass nucleus を直接含めていない。	新しいデバイスには、すべてのコンポーネントに関する依存関係の要件を満たす nucleus の最新パッチバージョンをインストールします。

[アクション]	デプロイ設定	nucleus の更新動作
	デプロイは、AWS が提供するコンポーネントを少なくとも 1 つ直接含み、あるいは AWS が提供するコンポーネントに依存するカスタムコンポーネント、または Greengrass nucleus のカスタムコンポーネントが含まれます。	既存のデバイスには、nucleus のインストールされたバージョンを更新しません。
デプロイを改訂せずに、既存のデプロイがターゲットするモノグループに新しいデバイスを追加します。	デプロイは、Greengrass nucleus の特定バージョンが直接含まれています。	新しいデバイスには、指定された nucleus バージョンをインストールします。 既存のデバイスには、nucleus のインストールされたバージョンを更新しません。
新しいデプロイを作成、あるいは既存のデプロイを修正します。	デプロイは Greengrass nucleus を直接含めていない。 デプロイは、AWS が提供するコンポーネントを少なくとも 1 つ直接含み、あるいは AWS が提供するコンポーネントに依存するカスタムコンポーネント、または Greengrass nucleus のカスタムコンポーネントが含まれます。	すべてのターゲットデバイスには、ターゲットのモノグループに追加する新しいデバイスをすべて含め、すべてのコンポーネントにおける依存関係の要件を満たす nucleus の最新パッチバージョンをインストールします。

[アクション]	デプロイ設定	nucleus の更新動作
新しいデプロイを作成、あるいは既存のデプロイを修正します。	デプロイは、Greengrass nucleus の特定バージョンが直接含まれています。	すべてのターゲットデバイスには、ターゲットのモノグループに追加する新しいデバイスをすべて含め、指定された nucleus バージョンをインストールします。

OTA 更新の実行

OTA 更新を実行するには、nucleus コンポーネントとインストールするバージョンを含むデプロイを作成します。

AWS IoT Greengrass Core ソフトウェアをアンインストールする

AWS IoT Greengrass Core ソフトウェアをアンインストールして、Greengrass コアデバイスとして使用しないデバイスからソフトウェアを削除できます。これらの手順を使用して、失敗したインストールをクリーンアップすることもできます。

AWS IoT Greengrass Core ソフトウェアをアンインストールするには

1. ソフトウェアをシステムサービスとして実行する場合は、サービスを停止、無効化、および削除する必要があります。オペレーティングシステムに適した次のコマンドを実行します。

   Linux

   1. サービスを停止します。

      ```
      sudo systemctl stop greengrass.service
      ```

   2. サービスを無効にします。

      ```
      sudo systemctl disable greengrass.service
      ```

   3. サービスを削除します。

```
sudo rm /etc/systemd/system/greengrass.service
```

4. サービスが削除されていることを確認します。

```
sudo systemctl daemon-reload && sudo systemctl reset-failed
```

Windows (Command Prompt)

> **Note**
>
> これらのコマンドを実行するには、管理者としてコマンドプロンプトを実行する必要があります。

1. サービスを停止します。

```
sc stop "greengrass"
```

2. サービスを無効にします。

```
sc config "greengrass" start=disabled
```

3. サービスを削除します。

```
sc delete "greengrass"
```

4. デバイスを再起動します。

Windows (PowerShell)

> **Note**
>
> これらのコマンドを実行するには、管理者 PowerShell として を実行する必要があります。

1. サービスを停止します。

```
Stop-Service -Name "greengrass"
```

2. サービスを無効にします。

```
Set-Service -Name "greengrass" -Status stopped -StartupType disabled
```

3. サービスを削除します。

   - PowerShell 6.0 以降の場合：

```
Remove-Service -Name "greengrass" -Confirm:$false -Verbose
```

   - 6.0 より前の PowerShell バージョンの場合：

```
Get-Item HKLM:\SYSTEM\CurrentControlSet\Services\greengrass | Remove-Item
 -Force -Verbose
```

4. デバイスを再起動します。

2. デバイスからルートフォルダを削除します。*/greengrass/v2* または *C:\greengrass\v2* をルートフォルダへのパスに置き換えます。

   Linux

```
sudo rm -rf /greengrass/v2
```

   Windows (Command Prompt)

```
rmdir /s /q C:\greengrass\v2
```

   Windows (PowerShell)

```
cmd.exe /c "rmdir /s /q C:\greengrass\v2"
```

3. AWS IoT Greengrass サービスからコアデバイスを削除します。この手順により、コアデバイスのステータス情報が AWS クラウド から削除されます。同じ名前のコアデバイスに AWS IoT Greengrass Core ソフトウェアを再インストールする予定の場合は、必ずこの手順を完了させてください。

   - AWS IoT Greengrass コンソールからコアデバイスを削除するには、次の手順を実行します。

a. AWS IoT Greengrass コンソールに移動します。
b. [Core devices] (コアデバイス) を選択します。
c. 削除するコアデバイスを選択します。
d. [削除] を選択します。
e. 確認モーダルで、[[Delete] (削除) を選択します。

- を使用してコアデバイスを削除するにはAWS Command Line Interface、DeleteCoreDeviceオペレーションを使用します。次のコマンドを実行し、 をコアデバイスの名前*MyGreengrassCore*に置き換えます。

```
aws greengrassv2 delete-core-device --core-device-thing-name MyGreengrassCore
```

AWS IoT Greengrass V2 のチュートリアル

次のチュートリアルを完了して、AWS IoT Greengrass V2 とその機能について学習できます。

トピック

チュートリアル: コンポーネントの更新を延期する Greengrass コンポーネントを開発する

このチュートリアルを完了すると、 over-the-air デプロイの更新を延期するコンポーネントを開発できます。デバイスに更新をデプロイするときに、次のような条件に基づいて更新を遅らせる必要が生じる場合があります。

- デバイスのバッテリー残量が少ない。
- デバイスが中断できないプロセスかジョブを実行中である。
- デバイスのインターネット接続が制限されているか、コストが高い。

> **Note**
>
> コンポーネントとは、AWS IoT Greengrass コアデバイスで実行されるソフトウェアモジュールです。コンポーネントを使用すると、複雑なアプリケーションを個別のビルディングブロックとして作成し管理することができ、Greengrass コアデバイス間で再利用することができます。

このチュートリアルでは、以下の作業を行います。

1. Greengrass Development Kit CLI (GDK CLI) を開発用コンピュータにインストールします。GDK CLI は、カスタムGreengrass コンポーネントの開発に役立つ機能を提供します。
2. コアデバイスのバッテリレベルがしきい値を下回ったときに、コンポーネントの更新を延期する Hello World コンポーネントを開発します。このコンポーネントは、SubscribeToComponentUpdatesIPC オペレーションを使用して更新通知をサブスクライブします。通知を受信すると、バッテリ残量がカスタマイズ可能なしきい値より低いかどうかを確認します。バッテリー残量がしきい値を下回っている場合、DeferComponentUpdateIPC オペレーションを使用して更新を 30 秒間延期します。このコンポーネントは、GDK CLI を使用して開発用コンピュータで開発します。

   > **Note**
   >
   > このコンポーネントは、実際のバッテリを模倣するために、コアデバイスで作成したファイルからバッテリレベルを読み取ります。そのため、このチュートリアルはバッテリのないコアデバイスでも実行できます。

3. このコンポーネントを AWS IoT Greengrass サービスにパブリッシュします。
4. このコンポーネントを、AWS クラウド からテスト先となる Greengrass コアデバイスにデプロイします。次に、コアデバイスの仮想バッテリレベルを変更し、別のデプロイを作成して、バッテリレベルが低いときにコアデバイスが更新を延期する様子を確認します。

このチュートリアルは 20～30 分を要します。

前提条件

このチュートリアルを完了するには、以下が必要です。

- AWS アカウント。アカウントをお持ちでない場合は、「AWS アカウント のセットアップ」を参照してください。
- 管理者権限を持つ AWS Identity and Access Management (IAM) ユーザー。
- インターネット接続が可能な Greengrass コアデバイス。コアデバイスの設定方法の詳細については、「AWS IoT Greengrass コアデバイスをセットアップする」を参照してください。
  - デバイス上のすべてのユーザーに Python 3.6 以降がインストールされており、PATH 環境変数に追加されていること。Windows の場合は、すべてのユーザーに対して Python Launcher for Windows ランチャーがインストールされている必要があります。

> **Important**
>
> Windows では、デフォルトでは Python がすべてのユーザーにインストールされません。Python をインストールするときには、インストールをカスタマイズして、AWS IoT Greengrass Core ソフトウェアが Python スクリプトを実行するように設定する必要があります。たとえば、グラフィカル Python インストーラを使用する場合には、次の操作を行います。
>
> 1. Install launcher for all users (recommended) (すべてのユーザーにランチャーをインストールする (推奨)) を選択します。
> 2. Customize installation を選択します。
> 3. を選択します。。Next
> 4. Install for all users を選択します。
> 5. Add Python to environment variables を選択します。
> 6. [Install] (インストール) を選択します。
>
> 詳細については、「Python 3 ドキュメント」の「Windows で Python を使用する」を参照してください。

- インターネットに接続された Windows、macOS、または Unix のような開発用コンピュータ。
  - Python 3.6 以降が開発コンピュータにインストールされていること。
  - Git が開発コンピュータにインストールされていること。
  - 開発コンピュータに AWS Command Line Interface (AWS CLI) がインストールされており、認証情報が設定されていること。詳細については、「AWS Command Line Interface ユーザーガイド」の「AWS CLI のインストール、更新、アンインストール」と「AWS CLI の設定」を参照してください。

> **Note**
>
> Raspberry Pi または別の 32 ビット ARM デバイスを使用する場合は、AWS CLI V1.AWS CLI をインストールします。V2 は 32 ビット ARM デバイスでは利用できません。詳細については、「AWS CLI バージョン 1 のインストール、更新、およびアンインストール」を参照してください。

ステップ 1: Greengrass Development Kit CLI をインストールする

Greengrass Development Kit CLI (GDK CLI) は、カスタム Greengrass コンポーネントの開発に役立つ機能を提供します。GDK CLI を使用して、カスタムコンポーネントを作成、ビルドおよびパブリッシュできます。

GDK CLI を開発コンピュータにインストールしていない場合は、次の手順を実行してインストールします。

GDK CLI の最新バージョンをインストールする

1. 開発コンピュータで次のコマンドを実行して、GitHubリポジトリ から GDK CLI の最新バージョンをインストールします。

   ```
   python3 -m pip install -U git+https://github.com/aws-greengrass/aws-greengrass-gdk-cli.git@v1.6.2
   ```

2. GDK CLI が正常にインストールされたことを確認するには、次のコマンドを実行します。

   ```
   gdk --help
   ```

   gdk コマンドが見つからない場合は、コマンドのフォルダを PATH に追加してください。

   - Linux デバイスで、 /home/*MyUser*/.local/binを PATH に追加し、 をユーザーの名前*MyUser*に置き換えます。
   - Windows デバイスでは、PATH *PythonPath*\\Scriptsに を追加し、 をデバイス上の Python フォルダへのパス*PythonPath*に置き換えます。

ステップ 2: 更新を延期するコンポーネントを開発する

このセクションでは、コアデバイスのバッテリレベルが、コンポーネントをデプロイしたときに設定したしきい値を下回ったときに、コンポーネントの更新を延期する Hello World コンポーネントを Python で開発します。このコンポーネントでは、AWS IoT Device SDK v2 for Python のプロセス間通信 (IPC) インターフェイスを使用します。SubscribeToComponentUpdates IPC オペレーションを使用して、コアデバイスがデプロイを受信したときに通知を受け取ります。次に、DeferComponentUpdateIPC オペレーションを使用して、デバイスのバッテリーレベルに基づいて更新を延期または承認します。

更新を延期する Hello World コンポーネントを開発するには

1. 開発コンピュータ上に、コンポーネントのソースコード用のフォルダを作成します。

```
mkdir com.example.BatteryAwareHelloWorld
cd com.example.BatteryAwareHelloWorld
```

2. テキストエディタを使用して gdk-config.json という名前のファイルを作成します。GDK CLI は gdk-config.json という名の GDK CLI 設定ファイルから読み込むことで、コンポーネントをビルドおよびパブリッシュします。この設定ファイルは、コンポーネントフォルダのルートにあります。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano gdk-config.json
```

   次の JSON をファイルにコピーします。

   - *Amazon* を自分の名前に置き換えます。
   - *us-west-2* をコアデバイスが動作している場所である AWS リージョン に置き換えます。GDK CLI がこのコンポーネントをこの AWS リージョン にパブリッシュします。
   - を使用する S3 バケットプレフィックス*greengrass-component-artifacts*に置き換えます。GDK CLI を使用してコンポーネントを公開すると、GDK CLI はコンポーネントの成果物を、この値、AWS リージョン および自分の AWS アカウント ID から形成される名前を持つ S3 バケットに、次のフォーマットを使用してアップロードします: *bucketPrefix*-*region*-*accountId*。

     例えば、**greengrass-component-artifacts** および **us-west-2** を指定し、AWS アカウント ID が **123456789012** の場合、GDK CLI は greengrass-component-artifacts-us-west-2-123456789012 という名前の S3 バケットを使用します。

```
{
  "component": {
    "com.example.BatteryAwareHelloWorld": {
      "author": "Amazon",
      "version": "NEXT_PATCH",
      "build": {
```

```
        "build_system" : "zip"
      },
      "publish": {
        "region": "us-west-2",
        "bucket": "greengrass-component-artifacts"
      }
    }
  },
  "gdk_version": "1.0.0"
}
```

設定ファイルは、以下を指定します。

- GDK CLI が Greengrass コンポーネントを AWS IoT Greengrass クラウドサービスにパブリッシュする際に使用するバージョン。NEXT_PATCH は、AWS IoT Greengrass クラウドサービスで利用可能な最新バージョンの次のパッチバージョンを選択するように指定します。コンポーネントに AWS IoT Greengrass クラウドサービスのバージョンがまだない場合は、GDK CLIは `1.0.0` を使用します。
- コンポーネントのビルドシステム。`zip` ビルドシステムを使用した場合、GDK CLI はコンポーネントのソースを ZIP ファイルにパッケージ化して、コンポーネントの単一のアーティファクトにします。
- GDK CLI が Greengrass コンポーネントをパブリッシュする場所となる AWS リージョン。
- GDK CLI がコンポーネントのアーティファクトをアップロードする場所となる S3 バケットのプレフィックス。

3. テキストエディタを使用して、`main.py` という名前のファイルにコンポーネントのソースコードを作成します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

   ```
   nano main.py
   ```

   ファイルに次の Python コードをコピーします。

   ```
   import json
   import os
   import sys
   import time
   import traceback
   ```

```
from pathlib import Path

from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2

HELLO_WORLD_PRINT_INTERVAL = 15  # Seconds
DEFER_COMPONENT_UPDATE_INTERVAL = 30 * 1000  # Milliseconds


class BatteryAwareHelloWorldPrinter():
    def __init__(self, ipc_client: GreengrassCoreIPCClientV2, battery_file_path:
 Path, battery_threshold: float):
        self.battery_file_path = battery_file_path
        self.battery_threshold = battery_threshold
        self.ipc_client = ipc_client
        self.subscription_operation = None

    def on_component_update_event(self, event):
        try:
            if event.pre_update_event is not None:
                if self.is_battery_below_threshold():
                    self.defer_update(event.pre_update_event.deployment_id)
                    print('Deferred update for deployment %s' %
                          event.pre_update_event.deployment_id)
                else:
                    self.acknowledge_update(
                        event.pre_update_event.deployment_id)
                    print('Acknowledged update for deployment %s' %
                          event.pre_update_event.deployment_id)
            elif event.post_update_event is not None:
                print('Applied update for deployment')
        except:
            traceback.print_exc()

    def subscribe_to_component_updates(self):
        if self.subscription_operation == None:
            # SubscribeToComponentUpdates returns a tuple with the response and the
 operation.
            _, self.subscription_operation =
 self.ipc_client.subscribe_to_component_updates(
                on_stream_event=self.on_component_update_event)

    def close_subscription(self):
        if self.subscription_operation is not None:
```

```
            self.subscription_operation.close()
            self.subscription_operation = None

    def defer_update(self, deployment_id):
        self.ipc_client.defer_component_update(
            deployment_id=deployment_id,
 recheck_after_ms=DEFER_COMPONENT_UPDATE_INTERVAL)

    def acknowledge_update(self, deployment_id):
        # Specify recheck_after_ms=0 to acknowledge a component update.
        self.ipc_client.defer_component_update(
            deployment_id=deployment_id, recheck_after_ms=0)

    def is_battery_below_threshold(self):
        return self.get_battery_level() < self.battery_threshold

    def get_battery_level(self):
        # Read the battery level from the virtual battery level file.
        with self.battery_file_path.open('r') as f:
            data = json.load(f)
            return float(data['battery_level'])

    def print_message(self):
        message = 'Hello, World!'
        if self.is_battery_below_threshold():
            message += ' Battery level (%d) is below threshold (%d), so the
 component will defer updates' % (
                self.get_battery_level(), self.battery_threshold)
        else:
            message += ' Battery level (%d) is above threshold (%d), so the
 component will acknowledge updates' % (
                self.get_battery_level(), self.battery_threshold)
        print(message)


def main():
    # Read the battery threshold and virtual battery file path from command-line
 args.
    args = sys.argv[1:]
    battery_threshold = float(args[0])
    battery_file_path = Path(args[1])
    print('Reading battery level from %s and deferring updates when below %d' % (
        str(battery_file_path), battery_threshold))
```

```
    try:
        # Create an IPC client and a Hello World printer that defers component
updates.
        ipc_client = GreengrassCoreIPCClientV2()
        hello_world_printer = BatteryAwareHelloWorldPrinter(
            ipc_client, battery_file_path, battery_threshold)
        hello_world_printer.subscribe_to_component_updates()
        try:
            # Keep the main thread alive, or the process will exit.
            while True:
                hello_world_printer.print_message()
                time.sleep(HELLO_WORLD_PRINT_INTERVAL)
        except InterruptedError:
            print('Subscription interrupted')
        hello_world_printer.close_subscription()
    except Exception:
        print('Exception occurred', file=sys.stderr)
        traceback.print_exc()
        exit(1)


if __name__ == '__main__':
    main()
```

この Python アプリケーションは次を実行します。

- 後でコアデバイスで作成する仮想バッテリレベルファイルから、コアデバイスのバッテリレベルを読み取ります。この仮想バッテリーレベルファイルは実際のバッテリーを模倣しているため、バッテリーのないコアデバイスでもこのチュートリアルを完了できます。
- バッテリのしきい値と、仮想バッテリレベルファイルへのパスに対するコマンドライン引数を読み取ります。コンポーネント recipe は、設定パラメータに基づいてこれらのコマンドライン引数を設定するため、コンポーネントをデプロイするときにこれらの値はカスタマイズできます。
- AWS IoT Device SDK v2 for Python で IPC クライアント V2 を使用して、AWS IoT Greengrass Core ソフトウェアと通信します。オリジナルの IPC クライアントと比較して、IPC クライアント V2で は、カスタムコンポーネントで IPC を使用するために記述する必要があるコードの量が減っています。
- SubscribeToComponentUpdates IPC オペレーションを使用して更新通知をサブスクライブします。AWS IoT Greengrass Core ソフトウェアは、各デプロイの前後に通知を送信します。コンポーネントは、通知を受信するたびに次の関数を呼び出します。通知が今後の

デプロイに関するものである場合、コンポーネントはバッテリ残量がしきい値より低いかどうかを確認します。バッテリー残量がしきい値を下回っている場合、コンポーネントは DeferComponentUpdate IPC オペレーションを使用して更新を 30 秒間延期します。それ以外の、バッテリ残量がしきい値を下回っていない場合には、コンポーネントは更新を承認し、更新が続行されます。

```
def on_component_update_event(self, event):
    try:
        if event.pre_update_event is not None:
            if self.is_battery_below_threshold():
                self.defer_update(event.pre_update_event.deployment_id)
                print('Deferred update for deployment %s' %
                      event.pre_update_event.deployment_id)
            else:
                self.acknowledge_update(
                    event.pre_update_event.deployment_id)
                print('Acknowledged update for deployment %s' %
                      event.pre_update_event.deployment_id)
        elif event.post_update_event is not None:
            print('Applied update for deployment')
    except:
        traceback.print_exc()
```

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアは、ローカルデプロイに関する更新通知は送信しないため、このコンポーネントは AWS IoT Greengrass クラウドサービスを使ってデプロイしてテストします。

4. テキストエディタを使用して、 `recipe.json` または `recipe.yaml` という名前のファイルにコンポーネント recipe を作成します。コンポーネント recipe は、コンポーネントのメタデータ、デフォルト設定パラメータ、プラットフォームに固有のライフサイクルスクリプトを定義します。

   JSON

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano recipe.json
```

次の JSON をファイルにコピーします。

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "COMPONENT_NAME",
  "ComponentVersion": "COMPONENT_VERSION",
  "ComponentDescription": "This Hello World component defers updates when the
 battery level is below a threshold.",
  "ComponentPublisher": "COMPONENT_AUTHOR",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "BatteryThreshold": 50,
      "LinuxBatteryFilePath": "/home/ggc_user/virtual_battery.json",
      "WindowsBatteryFilePath": "C:\\Users\\ggc_user\\virtual_battery.json"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "python3 -m pip install --user awsiotsdk --upgrade",
        "run": "python3 -u {artifacts:decompressedPath}/
com.example.BatteryAwareHelloWorld/main.py \"{configuration:/BatteryThreshold}\"
 \"{configuration:/LinuxBatteryFilePath}\""
      },
      "Artifacts": [
        {
          "Uri": "s3://BUCKET_NAME/COMPONENT_NAME/COMPONENT_VERSION/
com.example.BatteryAwareHelloWorld.zip",
          "Unarchive": "ZIP"
        }
      ]
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
```

```
        "install": "py -3 -m pip install --user awsiotsdk --upgrade",
        "run": "py -3 -u {artifacts:decompressedPath}/
com.example.BatteryAwareHelloWorld/main.py \"{configuration:/BatteryThreshold}\"
 \"{configuration:/WindowsBatteryFilePath}\""
      },
      "Artifacts": [
        {
          "Uri": "s3://BUCKET_NAME/COMPONENT_NAME/COMPONENT_VERSION/
com.example.BatteryAwareHelloWorld.zip",
          "Unarchive": "ZIP"
        }
      ]
    }
  ]
}
```

YAML

例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano recipe.yaml
```

ファイルに次の YAML をコピーします。

```
---
RecipeFormatVersion: "2020-01-25"
ComponentName: "COMPONENT_NAME"
ComponentVersion: "COMPONENT_VERSION"
ComponentDescription: "This Hello World component defers updates when the
 battery level is below a threshold."
ComponentPublisher: "COMPONENT_AUTHOR"
ComponentConfiguration:
  DefaultConfiguration:
    BatteryThreshold: 50
    LinuxBatteryFilePath: "/home/ggc_user/virtual_battery.json"
    WindowsBatteryFilePath: "C:\\Users\\ggc_user\\virtual_battery.json"
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: python3 -m pip install --user awsiotsdk --upgrade
```

```
      run: python3 -u {artifacts:decompressedPath}/
com.example.BatteryAwareHelloWorld/main.py "{configuration:/BatteryThreshold}"
 "{configuration:/LinuxBatteryFilePath}"
    Artifacts:
      - Uri: "s3://BUCKET_NAME/COMPONENT_NAME/COMPONENT_VERSION/
com.example.BatteryAwareHelloWorld.zip"
        Unarchive: ZIP
  - Platform:
      os: windows
    Lifecycle:
      install: py -3 -m pip install --user awsiotsdk --upgrade
      run: py -3 -u {artifacts:decompressedPath}/
com.example.BatteryAwareHelloWorld/main.py "{configuration:/BatteryThreshold}"
 "{configuration:/WindowsBatteryFilePath}"
    Artifacts:
      - Uri: "s3://BUCKET_NAME/COMPONENT_NAME/COMPONENT_VERSION/
com.example.BatteryAwareHelloWorld.zip"
        Unarchive: ZIP
```

この recipe は以下を指定します。

- バッテリのしきい値、Linux コアデバイスの仮想バッテリファイルパス、および Windows コアデバイスの仮想バッテリファイルパスに対するデフォルト設定パラメータ。
- AWS IoT Device SDK v2 for Python の最新バージョンをインストールする `install` ライフサイクル。
- `main.py` で Python アプリケーションを実行する `run` ライフサイクル。
- `COMPONENT_NAME` および `COMPONENT_VERSION` などのプレースホルダ。GDK CLI がコンポーネント recipe を構築するときにこの情報を置き換えます。

コンポーネント recipe の詳細については、「AWS IoT Greengrass コンポーネントレシピのリファレンス」を参照してください。

ステップ 3: コンポーネントを AWS IoT Greengrass サービスにパブリッシュします。

このセクションでは、Hello World コンポーネントを AWS IoT Greengrass クラウドサービスにパブリッシュします。コンポーネントが AWS IoT Greengrass クラウドサービスで利用できるようになった後に、コアデバイスにデプロイできます。GDK CLI を使用して、開発コンピュータから AWS

IoT Greengrass クラウドサービスにコンポーネントをパブリッシュします。GDK CLI が、ユーザーに代わってコンポーネントの recipe とアーティファクトをアップロードします。

Hello World コンポーネントを AWS IoT Greengrass サービスにパブリッシュするには

1. GDK CLI を使用して次のコマンドを実行し、コンポーネントを構築します。コンポーネントビルドコマンド は、GDK CLI 設定ファイルに基づいて、recipe とアーティファクトを作成します。このプロセスでは、GDK CLI が、コンポーネントのソースコードが含まれる ZIP ファイルを作成します。

   ```
   gdk component build
   ```

   次の例に示すようなメッセージが表示されます。

   ```
   [2022-04-28 11:20:16] INFO - Getting project configuration from gdk-config.json
   [2022-04-28 11:20:16] INFO - Found component recipe file 'recipe.yaml' in the
    project directory.
   [2022-04-28 11:20:16] INFO - Building the component
    'com.example.BatteryAwareHelloWorld' with the given project configuration.
   [2022-04-28 11:20:16] INFO - Using 'zip' build system to build the component.
   [2022-04-28 11:20:16] WARNING - This component is identified as using 'zip' build
    system. If this is incorrect, please exit and specify custom build command in the
    'gdk-config.json'.
   [2022-04-28 11:20:16] INFO - Zipping source code files of the component.
   [2022-04-28 11:20:16] INFO - Copying over the build artifacts to the greengrass
    component artifacts build folder.
   [2022-04-28 11:20:16] INFO - Updating artifact URIs in the recipe.
   [2022-04-28 11:20:16] INFO - Creating component recipe in 'C:\Users\finthomp
   \greengrassv2\com.example.BatteryAwareHelloWorld\greengrass-build\recipes'.
   ```

2. 次のコマンドを実行して、AWS IoT Greengrass クラウドサービスにコンポーネントをパブリッシュします。コンポーネントパブリッシュコマンドは、S3 バケットにコンポーネントの ZIP ファイルのアーティファクトをアップロードします。その後、コンポーネント recipe 内の ZIP ファイルの S3 URI を更新し、recipe を AWS IoT Greengrass サービスにアップロードします。この際、GDK CLI は、Hello World コンポーネントのどのバージョンがすでに AWS IoT Greengrass クラウドサービスで提供されているかを確認し、そのバージョンの次のパッチバージョンを選択できるようにします。コンポーネントがまだない場合は、GDK CLI は `1.0.0` バージョンを使用します。

   ```
   gdk component publish
   ```

次の例に示すようなメッセージが表示されます。出力には、GDK CLI が作成したコンポーネントのバージョンが示されます。

```
[2022-04-28 11:20:29] INFO - Getting project configuration from gdk-config.json
[2022-04-28 11:20:29] INFO - Found component recipe file 'recipe.yaml' in the
 project directory.
[2022-04-28 11:20:29] INFO - Found credentials in shared credentials file: ~/.aws/
credentials
[2022-04-28 11:20:30] INFO - No private version of the component
 'com.example.BatteryAwareHelloWorld' exist in the account. Using '1.0.0' as the
 next version to create.
[2022-04-28 11:20:30] INFO - Publishing the component
 'com.example.BatteryAwareHelloWorld' with the given project configuration.
[2022-04-28 11:20:30] INFO - Uploading the component built artifacts to s3 bucket.
[2022-04-28 11:20:30] INFO - Uploading component artifacts to S3
 bucket: greengrass-component-artifacts-us-west-2-123456789012. If this is your
 first time using this bucket, add the 's3:GetObject' permission to each core
 device's token exchange role to allow it to download the component artifacts. For
 more information, see https://docs.aws.amazon.com/greengrass/v2/developerguide/
device-service-role.html.
[2022-04-28 11:20:30] INFO - Not creating an artifacts bucket as it already exists.
[2022-04-28 11:20:30] INFO - Updating the component recipe
 com.example.BatteryAwareHelloWorld-1.0.0.
[2022-04-28 11:20:31] INFO - Creating a new greengrass component
 com.example.BatteryAwareHelloWorld-1.0.0
[2022-04-28 11:20:31] INFO - Created private version '1.0.0' of the component in
 the account.'com.example.BatteryAwareHelloWorld'.
```

3. S3 バケット名を出力からコピーします。後ほど、このバケット名を使用して、コアデバイスがこのバケットからコンポーネントアーティファクトをダウンロードできるようにします。
4. (オプション) コンポーネントを AWS IoT Greengrass コンソールを使用して、正常にアップロードされたことを確認します。以下の操作を実行します。

   a. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Components] (コンポーネント) を選択します。

   b. [Components] (コンポーネント) ページで、[My components] (マイコンポーネント) タブを選択し、次に com.example.BatteryAwareHelloWorld を選択します。

      このページには、コンポーネントの recipe と、コンポーネントに関するその他の情報が表示されます。

5. コアデバイスが S3 バケット内のコンポーネントアーティファクトにアクセスすることを許可します。

   各コアデバイスは、AWS IoT とやり取りや AWS クラウドへのログ送信を可能にするコアデバイス IAM ロールを有しています。このデバイスロールは、デフォルトでは S3 バケットへのアクセスを許可しないため、コアデバイスが S3 バケットからコンポーネントアーティファクトを取得できるようにするポリシーを作成して、アタッチする必要があります。

   デバイスのロールで S3 バケットへのアクセスが既に許可されている場合は、このステップを省略できます。そうでない場合は、次に示す方法で、アクセスを許可する IAM ポリシーを作成し、ロールにアタッチします。

   a. [IAM console] (IAM コンソール) ナビゲーションメニューで、[Policies] (ポリシー) を選択し、[Create policy] (ポリシーの作成) を選択します。

   b. JSON タブで、プレースホルダーコンテンツを以下のポリシーに置き換えます。*greengrass-component-artifacts-us-west-2-123456789012* を、GDK CLI がコンポーネントのアーティファクトをアップロードした S3 バケットの名前に置き換えます。

      例えば、GDK CLI の設定ファイルで **greengrass-component-artifacts** および **us-west-2** を指定し、AWS アカウント ID が **123456789012** の場合、GDK CLI は greengrass-component-artifacts-us-west-2-123456789012 という名前の S3 バケットを使用します。

      ```
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Action": [
              "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::greengrass-component-artifacts-us-
      west-2-123456789012/*"
          }
        ]
      }
      ```

   c. [次へ] をクリックします。

d. [ポリシーの詳細セクション] で、[名前] に「**MyGreengrassV2ComponentArtifactPolicy**」と入力します。

e. [ポリシーの作成] を選択します。

f. [IAM console] (IAM コンソール) ナビゲーションメニューで、[Role] (ロール) をクリックし、コアデバイスのロールの名前を選択します。このロール名は、AWS IoT Greengrass Core ソフトウェアをインストールしたときに指定したものです。名前を指定していない場合、デフォルトで `GreengrassV2TokenExchangeRole` が設定されます。

g. [Permissions] (アクセス許可) タブを選択し、[Add permissions] (アクセス許可の追加) を選択してから、[Attach policies] (ポリシーの添付) を選択します。

h. [アクセス許可の追加] ページで、作成した `MyGreengrassV2ComponentArtifactPolicy` ポリシーの横にあるチェックボックスを選択し、[アクセス許可の追加] を選択します。

ステップ 4: デバイス上でコンポーネントをデプロイしてテストする

このセクションでは、コンポーネントをコアデバイスにデプロイして、その機能をテストします。コアデバイスでは、実際のバッテリーを模倣する仮想バッテリレベルファイルを作成します。次に、別のデプロイを作成し、コアデバイス上のコンポーネントログファイルを確認して、コンポーネントの更新の延期と承認を確認します。

更新を延期する Hello World コンポーネントをデプロイしてテストするには

1. テキストエディタを使用して、仮想バッテリレベルファイルを作成します。このファイルは実際のバッテリーを模倣しています。

   - Linux コアデバイスの場合は、`/home/ggc_user/virtual_battery.json` という名前のファイルを作成します。`sudo` の権限でテキストエディタを実行します。
   - Windows コアデバイスの場合は、`C:\Users\ggc_user\virtual_battery.json` という名前のファイルを作成します。管理者としてテキストエディタを実行します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

   ```
   sudo nano /home/ggc_user/virtual_battery.json
   ```

   次の JSON をファイルにコピーします。

```
{
  "battery_level": 50
}
```

2. Hello World コンポーネントをコアデバイスにデプロイします。以下の操作を実行します。

   a. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Components] (コンポーネント) を選択します。

   b. [Components] (コンポーネント) ページで、[My components] (マイコンポーネント) タブを選択し、次に com.example.BatteryAwareHelloWorld を選択します。

   c. com.example.BatteryAwareHelloWorld ページで、[Deploy] (デプロイ) を選択します。

   d. [Add to deployment] (デプロイに追加) で、改訂する既存のデプロイを選択するか、新しいデプロイを作成することを選択して、[Next] (次へ) を選択します。

   e. 新しいデプロイの作成を選択した場合、デプロイのターゲットコアデバイスまたはモノグループを選択します。リポジトリの [Specify target] (ターゲットを指定) ページの、[Deployment target] (ターゲットのデプロイ) で、コアデバイスまたはモノグループを選択し、[Next] (次へ) を選択します。

   f. [Select components] (コンポーネントを選択) ページで、com.example.BatteryAwareHelloWorld コンポーネントが選択されていることを確認し、[Next] (次) を選択します。

   g. [Configure components] (コンポーネントを設定) ページで、com.example.BatteryAwareHelloWorld を選択したら、次の操作を行います。

      i. [Configure component] (コンポーネントを設定) を選択します。

      ii. [com.example.BatteryAwareHelloWorld の設定] モーダルの [設定の更新] の下にある [マージする設定] に、次の設定更新を入力します。

      ```
      {
        "BatteryThreshold": 70
      }
      ```

      iii. [Confirm] (確認) を選択してモーダルを閉じ、次に [Next] (次) を選択します。

   h. [Confirm advanced settings] (詳細設定の確認) ページ内の [Deployment policies] (デプロイポリシー) セクションにある [Component update policy] (コンポーネントの更新ポリシー) で、[Notify components] (コンポーネントに通知) が選択されていることを確認します。

新しいデプロイを作成するときには、[Notify components] (コンポーネントに通知) がデフォルトで選択されています。

i. [Review] ページで、[デプロイ] を選択します。

デプロイに最大 1 分かかる場合があります。

3. AWS IoT Greengrass Core ソフトウェアは、コンポーネントプロセスからの標準出力を `logs` フォルダのログファイルに保存します。次のコマンドを実行し、Hello World コンポーネントが実行され、ステータスメッセージが表示されることを確認します。

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/com.example.BatteryAwareHelloWorld.log
```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\logs\com.example.BatteryAwareHelloWorld.log
```

PowerShell

```
gc C:\greengrass\v2\logs\com.example.BatteryAwareHelloWorld.log -Tail 10 -Wait
```

次の例に示すようなメッセージが表示されます。

```
Hello, World! Battery level (50) is below threshold (70), so the component will
 defer updates.
```

> **Note**
>
> ファイルが存在しない場合、デプロイがまだ完了していない可能性があります。ファイルが 30 秒以内に表示されない場合は、デプロイが失敗している可能性があります。これは、コアデバイスに S3 バケットからコンポーネントのアーティファクトをダウンロードする権限がない場合などに発生します。次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアのログファイルを確認します。このファイルは、Greengrass コアデバイスのデプロイサービスからのログが含まれます。

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\logs\greengrass.log
```

type コマンドは、ファイルのコンテンツを端末に書き込みます。このコマンドを複数回実行して、ファイル内の変更を確認してください。

PowerShell

```
gc C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
```

4. コアデバイスへの新しいデプロイを作成し、コンポーネントが更新を延期していることを確認します。以下の操作を実行します。

   a. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Deployments] (デプロイ) を選択します。

   b. 前のステップで、作成したか、修正したデプロイを選択します。

   c. [deployment] (デプロイ) ページで、[Revise] (修正) を選択します。

   d. [Revise deployment] (デプロイの改訂) モーダルで、[Revise deployment] (デプロイの改訂) を選択します。

   e. それぞれのステップで [Next] (次) を選んでから、[Deploy] (デプロイ) を選択します。

5. 次のコマンドを実行して、コンポーネントのログを再度表示し、更新が延期されていることを確認します。

   Linux or Unix

   ```
   sudo tail -f /greengrass/v2/logs/com.example.BatteryAwareHelloWorld.log
   ```

   Windows Command Prompt (CMD)

   ```
   type C:\greengrass\v2\logs\com.example.BatteryAwareHelloWorld.log
   ```

PowerShell

```
gc C:\greengrass\v2\logs\com.example.BatteryAwareHelloWorld.log -Tail 10 -Wait
```

次の例に示すようなメッセージが表示されます。コンポーネントは更新を 30 秒間延期するため、コンポーネントはこのメッセージを繰り返し出力します。

```
Deferred update for deployment 50722a95-a05f-4e2a-9414-da80103269aa.
```

6. テキストエディタを使用して仮想バッテリ残量ファイルを編集し、バッテリレベルをしきい値を超える値に変更して、デプロイが続行されるようにします。

   - Linux コアデバイスの場合は、/home/ggc_user/virtual_battery.json という名前のファイルを編集します。sudo の権限でテキストエディタを実行します。
   - Windows コアデバイスの場合は、C:\Users\ggc_user\virtual_battery.json という名前のファイルを編集します。管理者としてテキストエディタを実行します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

   ```
   sudo nano /home/ggc_user/virtual_battery.json
   ```

   バッテリー残量を 80 に変更します。

   ```
   {
     "battery_level": 80
   }
   ```

7. 次のコマンドを実行して、コンポーネントのログを再度表示し、更新が承認されていることを確認します。

   Linux or Unix

   ```
   sudo tail -f /greengrass/v2/logs/com.example.BatteryAwareHelloWorld.log
   ```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\logs\com.example.BatteryAwareHelloWorld.log
```

PowerShell

```
gc C:\greengrass\v2\logs\com.example.BatteryAwareHelloWorld.log -Tail 10 -Wait
```

次の例に示されているようなメッセージが表示されます。

```
Hello, World! Battery level (80) is above threshold (70), so the component will
 acknowledge updates.
Acknowledged update for deployment f9499eb2-4a40-40a7-86c1-c89887d859f1.
```

これで、このチュートリアルは終了です。Hello World コンポーネントが、コアデバイスのバッテリ残量に基づいて更新を延期または承認します。このチュートリアルで説明しているトピックの詳細については、以下を参照してください。

- AWS IoT Greengrass コンポーネントを開発する
- デバイスに AWS IoT Greengrass コンポーネントのデプロイ
- AWS IoT Device SDK を使用して Greengrass nucleus、その他のコンポーネント、および AWS IoT Core と通信する
- AWS IoT Greengrass Development Kit Command-Line Interface

チュートリアル: MQTT 経由でローカル IoT デバイスとやり取りする

このチュートリアルを完了すると、コアデバイスでローカル IoT デバイスとやり取りできるように設定できます。ローカル IoT デバイスはクライアントデバイスとよばれ、MQTT 経由でコアデバイスに接続されます。このチュートリアルでは、AWS IoT モノがクライアントデバイスとしてコアデバイスに接続するために、Cloud Discovery を使用するように設定します。Cloud Discovery を設定すると、クライアントデバイスはリクエストを AWS IoT Greengrass クラウドサービスに送信してコアデバイスを検出することができます。AWS IoT Greengrass からのレスポンスには、クライアントデバイスが検出するように設定したコアデバイスの接続情報と証明書が含まれています。その後、ク

ライアントデバイスはこの情報を使用して利用可能なコアデバイスに接続することができ、そこから MQTT を介して操作できるようになります。

このチュートリアルでは、以下の作業を行います。

1. 必要に応じて、コアデバイスの権限を確認して更新します。
2. クライアントデバイスをコアデバイスに関連付けて、Cloud Discovery を使用してコアデバイスを検出できるようにします。
3. Greengrass コンポーネントをコアデバイスにデプロイして、クライアントデバイスのサポートを有効にします。
4. クライアントデバイスをコアデバイスに接続し、AWS IoT Core クラウドサービスとの通信をテストします。
5. クライアントデバイスと通信する、カスタムの Greengrass コンポーネントを開発します。
6. クライアントデバイスの AWS IoT デバイスシャドウを操作する、カスタムの Greengrass コンポーネントを開発します。

このチュートリアルでは、単一のコアデバイスと単一のクライアントデバイスを使用します。チュートリアルに従って、複数のクライアントデバイスを接続してテストすることもできます。

このチュートリアルは 30～60 分を要します。

前提条件

このチュートリアルを完了するには、以下が必要です。

- AWS アカウント。アカウントをお持ちでない場合は、「AWS アカウント のセットアップ」を参照してください。
- 管理者権限を持つ AWS Identity and Access Management (IAM) ユーザー。
- Greengrass コアデバイス。コアデバイスの設定方法の詳細については、「AWS IoT Greengrass コアデバイスをセットアップする」を参照してください。
  - コアデバイスで、Greengrass nucleus v2.6.0 以降が実行されている必要があります このバージョンには、ローカルの公開/サブスクライブ通信でのワイルドカードのサポートと、クライアントでのデバイスシャドウに関するサポートが含まれています。

> **Note**
>
> クライアントデバイスのサポートには、Greengrass nucleus v2.2.0 以降が必要です。このチュートリアルでは、ローカルの公開/サブスクライブでの MQTT ワイルドカードのサポートや、クライアントでのデバイスシャドウのサポートなど、新しい機能について説明しています。これらの機能を使用するには、Greengrass nucleus v2.6.0 以降が必要です。

  - コアデバイスは、接続するクライアントデバイスと同じネットワーク上にある必要があります。
  - (オプション) カスタムの Greengrass コンポーネントを開発しているモジュールを完了するには、コアデバイスで Greengrass CLI を実行している必要があります。詳細については、「Greengrass CLI のインストール」を参照してください。
- このチュートリアルで、クライアントデバイスとして接続する AWS IoT モノ。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT リソースを作成する」を参照してください。
  - クライアントデバイスの AWS IoT ポリシーでは、`greengrass:Discover` 権限を許可する必要があります。詳細については、「クライアントデバイス向けの最低限の AWS IoT ポリシー」を参照してください。
  - クライアントデバイスは、コアデバイスと同じネットワーク上にある必要があります。
  - クライアントデバイスは Python 3 を実行する必要があります。
  - クライアントデバイスは Git を実行する必要があります。

ステップ 1: コアデバイスの AWS IoT ポリシーを確認および更新する

クライアントデバイスをサポートするには、コアデバイスの AWS IoT ポリシーで、次のアクセス権限が付与されている必要があります。

- `greengrass:PutCertificateAuthorities`
- `greengrass:VerifyClientDeviceIdentity`
- `greengrass:VerifyClientDeviceIoTCertificateAssociation`
- `greengrass:GetConnectivityInfo`
- `greengrass:UpdateConnectivityInfo` - (オプション) このアクセス許可は、コアデバイスのネットワーク接続に関する情報を AWS IoT Greengrass クラウドサービスに報告する IP 検出コンポーネントを使用するために必要です。

コアデバイスのための、これらのアクセス許可と AWS IoT ポリシーの詳細については、「データプレーンオペレーションの AWS IoT ポリシー」および「クライアントデバイスをサポートするための最低限の AWS IoT ポリシー」を参照してください。

このセクションでは、コアデバイスの AWS IoT ポリシーを確認し、必要なアクセス許可が不足している場合には追加します。AWS IoT Greengrass Core ソフトウェアインストーラを使用してリソースをプロビジョニングした場合、コアデバイスには、すべての AWS IoT Greengrass アクション (`greengrass:*`) へのアクセスを許可する AWS IoT ポリシーが存在します。この状態で、シャドウマネージャーコンポーネントを設定してデバイスシャドウを AWS IoT Core と同期させる場合には、AWS IoT ポリシーのみを更新する必要があります。それ以外の場合、このセクションはスキップできます。

コアデバイスの AWS IoT ポリシーを確認および更新するには

1. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Core devices] (コアデバイス) を選択します。
2. [Core devices] (コアデバイス) ページで、更新するコアデバイスを選択します。
3. コアデバイスの詳細ページで、コアデバイスの [Thing] (モノ) へのリンクを選択します。このリンクをクリックすると、AWS IoT コンソールの [thing details] (モノ詳細) ページが開きます。
4. [thing details] (モノ詳細) ページで、[Certificates] (証明書) を選択します。
5. [Certificates] (証明書) タブで、モノのアクティブな証明書を選択します。
6. [certificate details] (証明書詳細) ページで、[Policies] (ポリシー) を選択します。
7. [Policies] (ポリシー) タブで、確認して更新する AWS IoT ポリシーを選択します。コアデバイスのアクティブな証明書にアタッチされている任意のポリシーに、必要なアクセス許可を追加できます。

   > **Note**
   >
   > リソースのプロビジョニングに AWS IoT Greengrass Core ソフトウェアインストーラを使用している場合、2 つの AWS IoT ポリシーが存在します。存在する場合は、GreengrassV2IoTThingPolicy という名前のポリシーを選択することをお勧めします。クイックインストーラで作成するコアデバイスは、デフォルトでこのポリシー名を使用します。このポリシーにアクセス許可を追加すると、このポリシーを使用する他のコアデバイスにもこれらのアクセス許可が付与されます。

8. [policy overview] (ポリシーの概要) で、[Edit active version] (アクティブなバージョンの編集) を選択します。

9. 必要なアクセス許可についてポリシーを確認し、不足していれば必要なアクセス許可を追加します。
10. 新しいポリシーバージョンをアクティブなバージョンとして設定するには、[Policy version status] (ポリシーバージョンのステータス) で、[Set the edited version as the active version for this policy] (編集したバージョンをこのポリシーのアクティブバージョンとして設定) を選択します。
11. [Save as new version] (新しいバージョンとして保存) を選択します。

ステップ 2: クライアントデバイスのサポートを有効にする

クライアントデバイスが Cloud Discovery を使用してコアデバイスに接続するには、デバイスを関連付ける必要があります。クライアントデバイスをコアデバイスに関連付けると、そのクライアントデバイスが、接続に使用するコアデバイスの IP アドレスと証明書を取得できるようになります。

クライアントデバイスがコアデバイスに安全に接続し、Greengrass コンポーネントと AWS IoT Core と通信できるようにするには、コアデバイスに次の Greengrass コンポーネントをデプロイします。

- クライアントデバイス認証 (`aws.greengrass.clientdevices.Auth`)

  クライアントデバイス認証コンポーネントをデプロイして、クライアントデバイスを認証し、クライアントデバイスのアクションを承認します。このコンポーネントは、AWS IoT モノがコアデバイスに接続できるようにします。

  このコンポーネントを使用するには、いくつかの設定が必要です。クライアントデバイスのグループを指定して、MQTT を介した接続や通信などの各グループが実行することができる操作を指定する必要があります。詳細については、「クライアントデバイス認証コンポーネントの設定」を参照してください。
- MQTT 3.1.1 ブローカー (モケット) (`aws.greengrass.clientdevices.mqtt.Moquette`)

  軽量な MQTT ブローカーを実行するため、Moquette MQTT ブローカーコンポーネントをデプロイします。Moquette MQTT ブローカーは MQTT 3.1.1 に準拠しており、QoS 0、QoS 1、QoS 2、保持されたメッセージ、Last Will メッセージ、および永続サブスクリプションに対するローカルサポートが含まれます。

  使用するにあたり、このコンポーネントを設定する必要はありません。ただし、このコンポーネントが MQTT ブローカを操作するポートを設定することができます。デフォルトではポート 8883 が使用されます。

- MQTT ブリッジ (`aws.greengrass.clientdevices.mqtt.Bridge`)

  (オプション) MQTT ブリッジコンポーネントをデプロイして、クライアントデバイス (ローカル MQTT)、ローカルパブリッシュ/サブスクライブ、および AWS IoT Core MQTT 間でメッセージをリレーします。クライアントデバイスを AWS IoT Core と同期するようにこのコンポーネントを設定し、Greengrass コンポーネントからクライアントデバイスとやり取りします。

  このコンポーネントを使用するには、設定する必要があります。このコンポーネントがメッセージをリレーするトピックマッピングを指定する必要があります。詳細については、「MQTT ブリッジコンポーネントの設定」を参照してください。

- IP ディテクター (`aws.greengrass.clientdevices.IPDetector`)

  (オプション) IP 検出コンポーネントをデプロイして、コアデバイスの MQTT ブローカエンドポイントを AWS IoT Greengrass クラウドサービスに自動的に報告します。ルータがコアデバイスに MQTT ブローカポートを転送する場合など、複雑なネットワーク設定がある場合には、このコンポーネントを使用することはできません。

  使用するにあたり、このコンポーネントを設定する必要はありません。

このセクションでは、AWS IoT Greengrass コンソールを使用して、クライアントデバイスを関連付けて、クライアントデバイスコンポーネントをコアデバイスにデプロイします。

クライアントデバイスのサポートを有効にするには

1. AWS IoT Greengrass コンソールに移動します。
2. 左のナビゲーションメニューで、[Core devices] (コアデバイス) を選択します。
3. [Core devices] (コアデバイス) ページで、クライアントデバイスのサポートを有効にするコアデバイスを選択します。
4. [core device details] (コアデバイスの詳細) ページで、[Client devices] (クライアントデバイス) タブを選択します。
5. [Client devices] (クライアントデバイス) タブで、[Configure cloud discovery] (Cloud Discoveryを設定する) を選択します。

   [Configure core device discovery] (コアデバイスディスカバリを設定する) ページが開きます。このページでは、クライアントデバイスをコアデバイスに関連付けて、クライアントデバイスコンポーネントをデプロイすることができます。このページでは、[Step 1: Select target core devices] (ステップ 1: ターゲットコアデバイスを選択する) でコアデバイスを選択します。

> **Note**
>
> このページを使用して、モノグループのコアデバイスディスカバリを設定することもできます。このオプションを選択すると、モノグループ内のすべてのコアデバイスにクライアントデバイスコンポーネントをデプロイできます。ただし、このオプションを選択した場合は、デプロイを作成した後に、クライアントデバイスを各コアデバイスに手動で関連付ける必要があります。このチュートリアルでは、単一のコアデバイスを設定します。

6. [Step 2: Associate client devices] (ステップ 2: クライアントデバイスを関連付ける) では、クライアントデバイスの AWS IoT モノをコアデバイスに関連付けます。こうすることで、クライアントデバイスが Cloud Discovery を使用して、コアデバイスの接続情報と証明書を取得できるようになります。以下の操作を実行します。

   a. [Associate client devices] (クライアントデバイスを関連付ける) を選択します。
   b. [Associate client devices with core device] (クライアントデバイスをコアデバイスに関連付ける) モーダルに関連付ける AWS IoT モノの名前を入力します。
   c. [追加] を選択します。
   d. [関連付ける] を選択します。

7. [Step 3: Configure and deploy Greengrass components] (ステップ 3: Greengrass コンポーネントを設定およびデプロイする) では、コンポーネントをデプロイして、クライアントデバイスのサポートを有効にします。ターゲットコアデバイスに以前のデプロイが存在する場合、このページでそのデプロイが修正されます。そうでない場合は、このページがコアデバイスの新しいデプロイを作成します。クライアントデバイスコンポーネントを設定およびデプロイするには、次の手順を実行します。

   a. このチュートリアルを完了するには、コアデバイスが Greengrass nucleus v2.6.0 以降を実行している必要があります。これより前のバージョンがコアデバイスでを実行されている場合には、以下を実行します。

      i. ボックスを選択して aws.greengrass.Nucleus コンポーネントをデプロイします。
      ii. aws.greengrass.Nucleus コンポーネントで [Edit configuration] (設定を編集する) を選択します。
      iii. [Component version] (コンポーネントバージョン) で、バージョン 2.6.0 以降を選択します。
      iv. [確認] を選択します。

> **Note**
>
> Greengrass nucleus を以前のマイナーバージョンからアップグレードする場合で、この nucleus に依存する AWS 提供のコンポーネントをコアデバイスが実行している際には、AWS 提供のコンポーネントの方も、新しいバージョンに更新する必要があります。このチュートリアルの後半でデプロイを確認するときに、これらのコンポーネントのバージョンを設定できます。詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

b. aws.greengrass.clientdevices.Auth コンポーネントで [Edit configuration] (設定を編集する) を選択します。

c. クライアントデバイス認証コンポーネントの [Edit configuration] (設定を編集する) モーダルで、クライアントデバイスがコアデバイス上の MQTT ブローカをパブリッシュしサブスクライブできるようにする承認ポリシーを設定します。以下の操作を実行します。

   i. [Configuration] (設定) にある [Configuration to merge] (マージする設定) のコードブロックに、次の設定を入力します。この設定にはクライアントデバイスの認証ポリシーが含まれています。各デバイスグループの承認ポリシーは、クライアントデバイスがこれらのアクションを実行できるアクションセットとリソースを指定しています。

   - このポリシーでは、名前が `MyClientDevice` で始まるクライアントデバイスが、すべての MQTT トピックに接続し通信することを許可しています。*`MyClientDevice*`* をクライアントデバイスとして接続するAWS IoTモノの名前に置き換えます。クライアントデバイスの名前と一致する * ワイルドカードを使って、名前を指定することもできます。* ワイルドカードは、名前の末尾にくる必要があります。

     接続する 2 番目のクライアントデバイスがある場合は、*`MyOtherClientDevice*`* をそのクライアントデバイスの名前、またはそのクライアントデバイスの名前と一致するワイルドカードパターンに置き換えます。ない場合は、削除するか、この選択ルールにある `MyOtherClientDevice*` と一致する名前のクライアントデバイスの接続と通信を許可するセクションをそのまま残しておくことができます。

   - このポリシーでは `OR` 演算子を使用して、名前が `MyOtherClientDevice` で始まるクライアントデバイスが、すべての MQTT トピックに接続し通信できるように許可しています。この部分は選択ルールから削除するか、接続するクライアントデバイスに合わせて修正することができます。

- このポリシーは、クライアントデバイスがすべての MQTT トピックでパブリッシュしサブスクライブすることを許可しています。セキュリティのベストプラクティスに従うため、mqtt:publish および mqtt:subscribe 操作はクライアントデバイスが通信に使用するトピックの最小セットになるように制限してください。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice* OR
 thingName: MyOtherClientDevice*",
        "policyName": "MyClientDevicePolicy"
      }
    },
    "policies": {
      "MyClientDevicePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish to all
 topics.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to all
 topics.",
          "operations": [
            "mqtt:subscribe"
          ],
```

```
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

詳細については、「クライアントデバイス認証コンポーネントの設定」を参照してください。

ii. [確認] を選択します。

d. aws.greengrass.clientdevices.mqtt.Bridge コンポーネントで [Edit configuration] (設定を編集する) を選択します。

e. MQTT ブリッジコンポーネントの [Edit configuration] (設定を編集する) モーダルで、クライアントデバイスから AWS IoT Core に MQTT メッセージをリレーするトピックマッピングを設定します。以下の操作を実行します。

i. [Configuration to merge] (マージする設定) コードブロック内にある [Configuration] (設定) に、次の設定を入力します。この設定では、クライアントデバイスから AWS IoT Core クラウドサービスに、`clients/+/hello/world` トピックフィルターの MQTT メッセージをリレーするように指定しています。例えば、このトピックフィルターは、`clients/MyClientDevice1/hello/world` トピックに一致します。

```
{
  "mqttTopicMapping": {
    "HelloWorldIotCoreMapping": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

詳細については、「MQTT ブリッジコンポーネントの設定」を参照してください。

ii. [確認] を選択します。

8. [Review and deploy] (レビューとデプロイ) を選択して、このページで作成されるデプロイを確認します。

9. このリージョンの Greengrass サービスロールをこれまで設定したことがない場合、コンソールにモーダルが表示され、サービスロールが設定されます。クライアントデバイス認証コンポーネントは、このサービスロールを使用してクライアントデバイスの ID を検証し、IP 検出コンポーネントはこのサービスロールを使用してコアデバイスの接続情報を管理します。[Grant permissions] (アクセス許可の付与) を選択します。
10. [Review] (レビュー) ページで、[Deploy] (デプロイ) を選択してコアデバイスへのデプロイを開始します。
11. デプロイが成功したかどうかを確認するため、デプロイのステータスと、コアデバイスのログを確認します。コアデバイスでのデプロイのステータスを確認するには、デプロイ [Overview] (概要) にある [Target] (ターゲット) を選択できます。詳細については、次を参照してください。

    - デプロイのステータスを確認する
    - AWS IoT Greengrass ログのモニタリング

ステップ 3: クライアントデバイスに接続する

クライアントデバイスでは AWS IoT Device SDK を使用して、コアデバイスを検出して接続し、通信することができます。クライアントデバイスは AWS IoT モノである必要があります。詳細については、「AWS IoT Core デベロッパーガイド」の「モノのオブジェクトを作成する」を参照してください。

このセクションでは、AWS IoT Device SDK v2 for Python をインストールして、AWS IoT Device SDK から Greengrass 検出サンプルアプリケーションを実行します。

> **Note**
>
> AWS IoT Device SDK は、他のプログラミング言語でも利用できます。このチュートリアルでは AWS IoT Device SDK v2 for Python を使用していますが、ユースケースに合わせて他の SDK を試すことができます。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT デバイス SDK」を参照してください。

クライアントデバイスをコアデバイスに接続するには

1. AWS IoT Device SDK v2 for Python をクライアントデバイスとして接続する AWS IoT モノにダウンロードしてインストールします。

   クライアントデバイスで、次の操作を行います。

a. AWS IoT Device SDK v2 for Python のリポジトリをクローンしてダウンロードします。

```
git clone https://github.com/aws/aws-iot-device-sdk-python-v2.git
```

b. AWS IoT Device SDK v2 for Python をインストールします。

```
python3 -m pip install --user ./aws-iot-device-sdk-python-v2
```

2. AWS IoT Device SDK v2 for Python のサンプルフォルダに移動します。

```
cd aws-iot-device-sdk-python-v2/samples
```

3. サンプルの Greengrass 検出アプリケーションを実行します。このアプリケーションでは、クライアントデバイスのモノの名前、使用する MQTT トピックとメッセージ、および接続を認証して保護する証明書を指定する引数が必要です。次の例では、`clients/`*`MyClientDevice1`*`/hello/world` トピックに「Hello World」メッセージを送信します。

> **Note**
>
> このトピックは、この前に、AWS IoT Core にメッセージをリレーするために MQTT ブリッジを設定した際のものと同じです。

- *`MyClientDevice1`* をクライアントデバイスのモノの名前に置き換えます。
- *`~/certs/AmazonRootCA1.pem`* をクライアントデバイスの Amazon ルート CA 証明書へのパスに置き換えます。
- *`~/certs/device.pem.crt`* をクライアントデバイスのデバイス証明書へのパスに置き換えます。
- *`~/certs/private.pem.key`* をクライアントデバイスのプライベートキーファイルへのパスに置き換えます。
- *`us-east-1`* をクライアントデバイスとコアデバイスが動作する AWS リージョンに置き換えます。

```
python3 basic_discovery.py \\
  --thing_name MyClientDevice1 \\
  --topic 'clients/MyClientDevice1/hello/world' \\
  --message 'Hello World!' \\
```

```
  --ca_file ~/certs/AmazonRootCA1.pem \\
  --cert ~/certs/device.pem.crt \\
  --key ~/certs/private.pem.key \\
  --region us-east-1 \\
  --verbosity Warn
```

検出サンプルアプリケーションはメッセージを 10 回送信し、その後切断します。また、メッセージの公開先と同じトピックにサブスクライブします。出力にアプリケーションがトピックに関する MQTT メッセージを受信したことが示される場合は、クライアントデバイスはコアデバイスと正常に通信できています。

```
Performing greengrass discovery...
awsiot.greengrass_discovery.DiscoverResponse(gg_groups=[awsiot.greengrass_discovery.GGGroup
coreDevice-MyGreengrassCore',
 cores=[awsiot.greengrass_discovery.GGCore(thing_arn='arn:aws:iot:us-
east-1:123456789012:thing/MyGreengrassCore',
 connectivity=[awsiot.greengrass_discovery.ConnectivityInfo(id='203.0.113.0',
 host_address='203.0.113.0', metadata='', port=8883)])],
 certificate_authorities=['-----BEGIN CERTIFICATE-----\
MIICiT...EXAMPLE=\
-----END CERTIFICATE-----\
'])])
Trying core arn:aws:iot:us-east-1:123456789012:thing/MyGreengrassCore at host
 203.0.113.0 port 8883
Connected!
Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!",
 "sequence": 0}

Publish received on topic clients/MyClientDevice1/hello/world
b'{"message": "Hello World!", "sequence": 0}'
Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!",
 "sequence": 1}

Publish received on topic clients/MyClientDevice1/hello/world
b'{"message": "Hello World!", "sequence": 1}'

...

Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!",
 "sequence": 9}

Publish received on topic clients/MyClientDevice1/hello/world
```

```
b'{"message": "Hello World!", "sequence": 9}'
```

アプリケーションが代わりにエラーを出力する場合は、「Greengrass 検出で生じる問題のトラブルシューティング」を参照してください。

コアデバイスの Greengrass ログを表示して、クライアントデバイスが正常に接続してメッセージを送信しているかどうかを確認することもできます。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

4. MQTT ブリッジがクライアントデバイスからのメッセージをに AWS IoT Core リレーしていることを確認します。AWS IoT Core コンソールの MQTT テストクライアントを使用して、MQTT トピックフィルターにサブスクライブすることができます。以下の操作を実行します。

   a. AWS IoT コンソールに移動します。
   b. 左側のナビゲーションメニューの [Test] (テスト) で、[MQTT test client] (MQTT テストクライアント) を選択します。
   c. [Subscribe to a topic] (トピックへのサブスクライブ) タブの [Topic filter] (トピックのフィルター) に `clients/+/hello/world` と入力して、コアデバイスからクライアントデバイスメッセージをサブスクライブします。
   d. [サブスクライブ] を選択します。
   e. クライアントデバイスでパブリッシュ/サブスクライブアプリケーションを再度実行します。

      MQTT テストクライアントに、クライアントデバイスから送信したこのトピックフィルターに一致するトピックについてのメッセージが表示されます。

ステップ 4: クライアントデバイスと通信するコンポーネントを開発する

クライアントデバイスと通信する Greengrass コンポーネントを開発することができます。コンポーネントはプロセス間通信 (IPC) とローカルパブリッシュ/サブスクライブインターフェイスを使用して、コアデバイス上で通信します。クライアントデバイスとやり取りするには、クライアントデバイスとローカルの公開/サブスクライブインターフェース間でメッセージをリレーするように、MQTT ブリッジコンポーネントを設定します。

このセクションでは、クライアントデバイスからローカルのパブリッシュ/サブスクライブインターフェイスにメッセージをリレーするように、MQTT ブリッジコンポーネントを更新します。その後、これらのメッセージにサブスクライブし、メッセージを受信したときにメッセージを表示するコンポーネントを開発します。

クライアントデバイスと通信するコンポーネントを開発するには

1. コアデバイスへのデプロイを修正し、クライアントデバイスからローカルパブリッシュ/サブスクライブにメッセージをリレーするように MQTT ブリッジコンポーネントを設定します。以下の操作を実行します。

   a. AWS IoT Greengrass コンソールに移動します。

   b. 左のナビゲーションメニューで、[Core devices] (コアデバイス) を選択します。

   c. [Core devices] (コアデバイス) ページで、このチュートリアルで使用するコアデバイスを選択します。

   d. [core device details] (コアデバイスの詳細) ページで、[Client devices] (クライアントデバイス) タブを選択します。

   e. [Client devices] (クライアントデバイス) タブで、[Configure cloud discovery] (Cloud Discoveryを設定する) を選択します。

      [Configure core device discovery] (コアデバイスディスカバリを設定する) ページが開きます。このページでは、コアデバイスにデプロイするクライアントデバイスコンポーネントを変更または設定できます。

   f. [Step 3] (ステップ 3) の aws.greengrass.clientdevices.mqtt.Bridge コンポーネントでは、[Edit configuration] (設定を編集する) を選択します。

   g. MQTT ブリッジコンポーネントの [Edit configuration] (設定を編集する) モーダルで、クライアントデバイスからローカルのパブリッシュ/サブスクライブインターフェースに MQTT メッセージをリレーするトピックマッピングを設定します。以下の操作を実行します。

      i. [Configuration to merge] (マージする設定) コードブロック内にある [Configuration] (設定) に、次の設定を入力します。この設定では、クライアントデバイスから AWS IoT Core クラウドサービスおよびローカル Greengrass のパブリッシュ/サブスクライブブローカーに、`clients/+/hello/world` トピックフィルターと一致するトピックに関する MQTT メッセージをリレーするように指定しています。

```
{
  "mqttTopicMapping": {
    "HelloWorldIotCoreMapping": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
    "HelloWorldPubsubMapping": {
```

```
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "Pubsub"
    }
  }
}
```

詳細については、「MQTT ブリッジコンポーネントの設定」を参照してください。

ii. [確認] を選択します。

h. [Review and deploy] (レビューとデプロイ) を選択して、このページで作成されるデプロイを確認します。

i. [Review] (レビュー) ページで、[Deploy] (デプロイ) を選択してコアデバイスへのデプロイを開始します。

j. デプロイが成功したかどうかを確認するため、デプロイのステータスと、コアデバイスのログを確認します。コアデバイスでのデプロイのステータスを確認するには、デプロイ [Overview] (概要) にある [Target] (ターゲット) を選択できます。詳細については、次を参照してください。

- デプロイのステータスを確認する
- AWS IoT Greengrass ログのモニタリング

2. クライアントデバイスからの Hello World メッセージをサブスクライブする Greengrass コンポーネントを開発してデプロイします。以下の操作を実行します。

a. コアデバイスに、recipe とアーティファクト用のフォルダを作成します。

Linux or Unix

```
mkdir recipes
mkdir -p artifacts/com.example.clientdevices.MyHelloWorldSubscriber/1.0.0
```

Windows Command Prompt (CMD)

```
mkdir recipes
mkdir artifacts\com.example.clientdevices.MyHelloWorldSubscriber\1.0.0
```

PowerShell

```
mkdir recipes
```

```
mkdir artifacts\com.example.clientdevices.MyHelloWorldSubscriber\1.0.0
```

> **Important**
>
> アーティファクトフォルダのパスには、次のフォーマットを使用する必要があります。recipe で指定したコンポーネント名とバージョンを含めてください。
>
> ```
> artifacts/componentName/componentVersion/
> ```

b. テキストエディタを使用し、次の内容でコンポーネント recipe を作成します。この recipe では、AWS IoT Device SDK v2 for Python をインストールし、トピックにサブスクライブしてメッセージを出力するスクリプトを実行するように指定しています。

例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano recipes/com.example.clientdevices.MyHelloWorldSubscriber-1.0.0.json
```

ファイル内には、次の recipe をコピーします。

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.clientdevices.MyHelloWorldSubscriber",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to Hello World messages
 from client devices.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.clientdevices.MyHelloWorldSubscriber:pubsub:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "*"
            ]
```

```
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "python3 -m pip install --user awsiotsdk",
        "run": "python3 -u {artifacts:path}/hello_world_subscriber.py"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "py -3 -m pip install --user awsiotsdk",
        "run": "py -3 -u {artifacts:path}/hello_world_subscriber.py"
      }
    }
  ]
}
```

c. テキストエディタを使用し、hello_world_subscriber.py という名前の Python スクリプトアーティファクトを、次の内容で作成します。このアプリケーションは、パブリッシュ/サブスクライブ IPC サービスを使用して、clients/+/hello/world トピックにサブスクライブし、受信したメッセージを出力します。

例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano artifacts/com.example.clientdevices.MyHelloWorldSubscriber/1.0.0/
hello_world_subscriber.py
```

ファイルに次の Python コードをコピーします。

```
import sys
import time
```

```
import traceback

from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2

CLIENT_DEVICE_HELLO_WORLD_TOPIC = 'clients/+/hello/world'
TIMEOUT = 10


def on_hello_world_message(event):
    try:
        message = str(event.binary_message.message, 'utf-8')
        print('Received new message: %s' % message)
    except:
        traceback.print_exc()


try:
    ipc_client = GreengrassCoreIPCClientV2()

    # SubscribeToTopic returns a tuple with the response and the operation.
    _, operation = ipc_client.subscribe_to_topic(
        topic=CLIENT_DEVICE_HELLO_WORLD_TOPIC,
 on_stream_event=on_hello_world_message)
    print('Successfully subscribed to topic: %s' %
          CLIENT_DEVICE_HELLO_WORLD_TOPIC)

    # Keep the main thread alive, or the process will exit.
    try:
        while True:
            time.sleep(10)
    except InterruptedError:
        print('Subscribe interrupted.')

    operation.close()
except Exception:
    print('Exception occurred when using IPC.', file=sys.stderr)
    traceback.print_exc()
    exit(1)
```

> **Note**
>
> このコンポーネントでは、AWS IoT Device SDK v2 for Python の IPC クライアント V2 を使用して、AWS IoT Greengrass Core ソフトウェアと通信しています。オ

> リジナルの IPC クライアントと比較して、IPC クライアント V2で は、カスタムコンポーネントで IPC を使用するために記述する必要があるコードの量が減っています。

d. Greengrass CLI を使用してコンポーネントをデプロイします。

Linux or Unix

```
sudo /greengrass/v2/bin/greengrass-cli deployment create \
  --recipeDir recipes \
  --artifactDir artifacts \
  --merge "com.example.clientdevices.MyHelloWorldSubscriber=1.0.0"
```

Windows Command Prompt (CMD)

```
C:\greengrass\v2/bin/greengrass-cli deployment create ^
  --recipeDir recipes ^
  --artifactDir artifacts ^
  --merge "com.example.clientdevices.MyHelloWorldSubscriber=1.0.0"
```

PowerShell

```
C:\greengrass\v2/bin/greengrass-cli deployment create `
  --recipeDir recipes `
  --artifactDir artifacts `
  --merge "com.example.clientdevices.MyHelloWorldSubscriber=1.0.0"
```

3. コンポーネントログを表示して、コンポーネントが正常にインストールされ、トピックにサブスクライブされていることを確認します。

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/
com.example.clientdevices.MyHelloWorldSubscriber.log
```

PowerShell

```
gc C:\greengrass\v2/logs/com.example.clientdevices.MyHelloWorldSubscriber.log -
Tail 10 -Wait
```

コアデバイスがメッセージを受信しているかどうかを確認できるように、ログフィードは開いたままにしておくことができます。

4. クライアントデバイスで、サンプルの Greengrass 検出アプリケーションを再度実行して、コアデバイスにメッセージを送信します。

```
python3 basic_discovery.py \\
  --thing_name MyClientDevice1 \\
  --topic 'clients/MyClientDevice1/hello/world' \\
  --message 'Hello World!' \\
  --ca_file ~/certs/AmazonRootCA1.pem \\
  --cert ~/certs/device.pem.crt \\
  --key ~/certs/private.pem.key \\
  --region us-east-1 \\
  --verbosity Warn
```

5. コンポーネントログをもう一度表示して、コンポーネントがクライアントデバイスからのメッセージを受信して出力していることを確認します。

   Linux or Unix

   ```
   sudo tail -f /greengrass/v2/logs/
   com.example.clientdevices.MyHelloWorldSubscriber.log
   ```

   PowerShell

   ```
   gc C:\greengrass\v2/logs/com.example.clientdevices.MyHelloWorldSubscriber.log -
   Tail 10 -Wait
   ```

ステップ 5: クライアントのデバイスシャドウを操作するコンポーネントを開発する

クライアントデバイスの AWS IoT デバイスシャドウとやり取りする、Greengrass コンポーネントを開発することができます。シャドウは、AWS IoT でのモノ (クライアントデバイスなど) に関する、現在または所望の状態についての情報を保存する JSON ドキュメントです。カスタムコンポーネントは、クライアントデバイスが AWS IoT に接続されていない場合でも、クライアントデバイスのシャドウにアクセスしてその状態を管理できます。AWS IoT のモノには、それぞれに名前のないシャドウがあり、また、モノごとに、名前付きシャドウを複数作成することができます。

このセクションでは、シャドウマネージャーコンポーネントをデプロイし、コアデバイスのシャドウを管理します。また、クライアントデバイスとシャドウマネージャーコンポーネント間でシャドウメッセージをリレーするように、MQTT ブリッジコンポーネントを更新します。その後、クライアントデバイスのシャドウを更新するコンポーネントを開発し、コンポーネントからのシャドウの更新に応答するサンプルアプリケーションを、クライアントデバイスで実行します。このコンポーネントは、コアデバイスにクライアントデバイスとして接続された、スマートライトの色の状態を管理する、スマートライト管理アプリケーションのものです。

クライアントのデバイスシャドウを操作するコンポーネントを開発するには

1. シャドウマネージャーコンポーネントをデプロイするようにコアデバイスのデプロイを修正し、クライアントデバイスからローカルの公開/サブスクライブにシャドウメッセージをリレーするように、MQTT ブリッジコンポーネントを設定します。以下の操作を実行します。

   a. AWS IoT Greengrass コンソールに移動します。

   b. 左のナビゲーションメニューで、[Core devices] (コアデバイス) を選択します。

   c. [Core devices] (コアデバイス) ページで、このチュートリアルで使用するコアデバイスを選択します。

   d. [core device details] (コアデバイスの詳細) ページで、[Client devices] (クライアントデバイス) タブを選択します。

   e. [Client devices] (クライアントデバイス) タブで、[Configure cloud discovery] (Cloud Discoveryを設定する) を選択します。

      [Configure core device discovery] (コアデバイスディスカバリを設定する) ページが開きます。このページでは、コアデバイスにデプロイするクライアントデバイスコンポーネントを変更または設定できます。

   f. [Step 3] (ステップ 3) の aws.greengrass.clientdevices.mqtt.Bridge コンポーネントでは、[Edit configuration] (設定を編集する) を選択します。

   g. MQTT ブリッジコンポーネントの [Edit configuration] (設定を編集) モーダルで、クライアントデバイスとローカルの公開/サブスクライブインターフェース間で、デバイスシャドウトピックの MQTT メッセージを伝達するための、トピックマッピングを設定します。同時に、互換性のある MQTT ブリッジバージョンを、デプロイで指定されていることも確認します。クライアントでのデバイスシャドウのサポートには、MQTT ブリッジ v2.2.0 以降が必要です。以下の操作を実行します。

      i. [Component version] (コンポーネントバージョン) で、バージョン 2.2.0 以降を選択します。

ii. [Configuration to merge] (マージする設定) コードブロック内にある [Configuration] (設定) に、次の設定を入力します。この設定では、シャドウトピックで MQTT メッセージを伝達するように指定しています。

```
{
  "mqttTopicMapping": {
    "HelloWorldIotCoreMapping": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
    "HelloWorldPubsubMapping": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ShadowsLocalMqttToPubsub": {
      "topic": "$aws/things/+/shadow/#",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ShadowsPubsubToLocalMqtt": {
      "topic": "$aws/things/+/shadow/#",
      "source": "Pubsub",
      "target": "LocalMqtt"
    }
  }
}
```

詳細については、「MQTT ブリッジコンポーネントの設定」を参照してください。

iii. [確認] を選択します。

h. [Step 3] (ステップ 3) で、デプロイする aws.greengrass.ShadowManager コンポーネントを選択します。

i. [Review and deploy] (レビューとデプロイ) を選択して、このページで作成されるデプロイを確認します。

j. [Review] (レビュー) ページで、[Deploy] (デプロイ) を選択してコアデバイスへのデプロイを開始します。

k. デプロイが成功したかどうかを確認するため、デプロイのステータスと、コアデバイスのログを確認します。コアデバイスでのデプロイのステータスを確認するには、デプロイ

[Overview] (概要) にある [Target] (ターゲット) を選択できます。詳細については、次を参照してください。

- デプロイのステータスを確認する
- AWS IoT Greengrass ログのモニタリング

2. スマートライトのクライアントデバイスを管理する、Greengrass コンポーネントの開発とデプロイを行います。以下の操作を実行します。

   a. コンポーネントのアーティファクト用のフォルダーを、コアデバイスに作成します。

   Linux or Unix

   ```
   mkdir -p artifacts/com.example.clientdevices.MySmartLightManager/1.0.0
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir artifacts\com.example.clientdevices.MySmartLightManager\1.0.0
   ```

   PowerShell

   ```
   mkdir artifacts\com.example.clientdevices.MySmartLightManager\1.0.0
   ```

   > **Important**
   >
   > アーティファクトフォルダのパスには、次のフォーマットを使用する必要があります。recipe で指定したコンポーネント名とバージョンを含めてください。
   >
   > ```
   > artifacts/componentName/componentVersion/
   > ```

   b. テキストエディタを使用し、次の内容でコンポーネント recipe を作成します。この recipe では、AWS IoT Device SDK v2 for Python をインストールし、スマートライトのクライアントデバイスのシャドウを操作して色を管理するための、スクリプトを実行するように指定しています

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano recipes/com.example.clientdevices.MySmartLightManager-1.0.0.json
```

ファイル内には、次の recipe をコピーします。

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.clientdevices.MySmartLightManager",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that interacts with smart light client
 devices.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.Nucleus": {
      "VersionRequirement": "^2.6.0"
    },
    "aws.greengrass.ShadowManager": {
      "VersionRequirement": "^2.2.0"
    },
    "aws.greengrass.clientdevices.mqtt.Bridge": {
      "VersionRequirement": "^2.2.0"
    }
  },
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "smartLightDeviceNames": [],
      "accessControl": {
        "aws.greengrass.ShadowManager": {
          "com.example.clientdevices.MySmartLightManager:shadow:1": {
            "policyDescription": "Allows access to client devices' unnamed
 shadows",
            "operations": [
              "aws.greengrass#GetThingShadow",
              "aws.greengrass#UpdateThingShadow"
            ],
            "resources": [
              "$aws/things/MyClientDevice*/shadow"
            ]
          }
        },
        "aws.greengrass.ipc.pubsub": {
          "com.example.clientdevices.MySmartLightManager:pubsub:1": {
```

```
            "policyDescription": "Allows access to client devices' unnamed
 shadow updates",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "$aws/things/+/shadow/update/accepted"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "python3 -m pip install --user awsiotsdk",
        "run": "python3 -u {artifacts:path}/smart_light_manager.py"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "py -3 -m pip install --user awsiotsdk",
        "run": "py -3 -u {artifacts:path}/smart_light_manager.py"
      }
    }
  ]
}
```

c. テキストエディタを使用し、`smart_light_manager.py` という名前の Python スクリプトアーティファクトを、次の内容で作成します。このアプリケーションは、シャドウ IPC サービスを使用してクライアントデバイスのシャドウを取得および更新します。また、ローカルの公開/サブスクライブ用 IPC サービスを使用して、報告されたシャドウアップデートの受け取りも行います。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano artifacts/com.example.clientdevices.MySmartLightManager/1.0.0/
smart_light_manager.py
```

ファイルに次の Python コードをコピーします。

```
import json
import random
import sys
import time
import traceback
from uuid import uuid4

from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2
from awsiot.greengrasscoreipc.model import ResourceNotFoundError

SHADOW_COLOR_PROPERTY = 'color'
CONFIGURATION_CLIENT_DEVICE_NAMES = 'smartLightDeviceNames'
COLORS = ['red', 'orange', 'yellow', 'green', 'blue', 'purple']
SHADOW_UPDATE_TOPIC = '$aws/things/+/shadow/update/accepted'
SET_COLOR_INTERVAL = 15


class SmartLightDevice():
    def __init__(self, client_device_name: str, reported_color: str = None):
        self.name = client_device_name
        self.reported_color = reported_color
        self.desired_color = None


class SmartLightDeviceManager():
    def __init__(self, ipc_client: GreengrassCoreIPCClientV2):
        self.ipc_client = ipc_client
        self.devices = {}
        self.client_tokens = set()
        self.shadow_update_accepted_subscription_operation = None
        self.client_device_names_configuration_subscription_operation = None
        self.update_smart_light_device_list()

    def update_smart_light_device_list(self):
        # Update the device list from the component configuration.
        response = self.ipc_client.get_configuration(
            key_path=[CONFIGURATION_CLIENT_DEVICE_NAMES])
```

```
        # Identify the difference between the configuration and the currently
tracked devices.
        current_device_names = self.devices.keys()
        updated_device_names =
response.value[CONFIGURATION_CLIENT_DEVICE_NAMES]
        added_device_names = set(updated_device_names) -
set(current_device_names)
        removed_device_names = set(current_device_names) -
set(updated_device_names)
        # Stop tracking any smart light devices that are no longer in the
configuration.
        for name in removed_device_names:
            print('Removing %s from smart light device manager' % name)
            self.devices.pop(name)
        # Start tracking any new smart light devices that are in the
configuration.
        for name in added_device_names:
            print('Adding %s to smart light device manager' % name)
            device = SmartLightDevice(name)
            device.reported_color = self.get_device_reported_color(device)
            self.devices[name] = device
            print('Current color for %s is %s' % (name,
device.reported_color))

    def get_device_reported_color(self, smart_light_device):
        try:
            response = self.ipc_client.get_thing_shadow(
                thing_name=smart_light_device.name, shadow_name='')
            shadow = json.loads(str(response.payload, 'utf-8'))
            if 'reported' in shadow['state']:
                return shadow['state']['reported'].get(SHADOW_COLOR_PROPERTY)
            return None
        except ResourceNotFoundError:
            return None

    def request_device_color_change(self, smart_light_device, color):
        # Generate and track a client token for the request.
        client_token = str(uuid4())
        self.client_tokens.add(client_token)
        # Create a shadow payload, which must be a blob.
        payload_json = {
            'state': {
                'desired': {
                    SHADOW_COLOR_PROPERTY: color
```

```
                }
            },
            'clientToken': client_token
        }
        payload = bytes(json.dumps(payload_json), 'utf-8')
        self.ipc_client.update_thing_shadow(
            thing_name=smart_light_device.name, shadow_name='',
payload=payload)
        smart_light_device.desired_color = color

    def subscribe_to_shadow_update_accepted_events(self):
        if self.shadow_update_accepted_subscription_operation == None:
            # SubscribeToTopic returns a tuple with the response and the
operation.
            _, self.shadow_update_accepted_subscription_operation =
self.ipc_client.subscribe_to_topic(
                topic=SHADOW_UPDATE_TOPIC,
on_stream_event=self.on_shadow_update_accepted_event)
            print('Successfully subscribed to shadow update accepted topic')

    def close_shadow_update_accepted_subscription(self):
        if self.shadow_update_accepted_subscription_operation is not None:
            self.shadow_update_accepted_subscription_operation.close()

    def on_shadow_update_accepted_event(self, event):
        try:
            message = str(event.binary_message.message, 'utf-8')
            accepted_payload = json.loads(message)
            # Check for reported states from smart light devices and ignore
desired states from components.
            if 'reported' in accepted_payload['state']:
                # Process this update only if it uses a client token created by
this component.
                client_token = accepted_payload.get('clientToken')
                if client_token is not None and client_token in
self.client_tokens:
                    self.client_tokens.remove(client_token)
                    shadow_state = accepted_payload['state']['reported']
                    if SHADOW_COLOR_PROPERTY in shadow_state:
                        reported_color = shadow_state[SHADOW_COLOR_PROPERTY]
                        topic = event.binary_message.context.topic
                        client_device_name = topic.split('/')[2]
                        if client_device_name in self.devices:
```

```
                            # Set the reported color for the smart light
device.
                            self.devices[client_device_name].reported_color =
reported_color
                            print(
                                'Received shadow update confirmation from
client device: %s' % client_device_name)
                    else:
                        print("Shadow update doesn't specify color")
        except:
            traceback.print_exc()

    def subscribe_to_client_device_name_configuration_updates(self):
        if self.client_device_names_configuration_subscription_operation ==
None:
            # SubscribeToConfigurationUpdate returns a tuple with the response
and the operation.
            _, self.client_device_names_configuration_subscription_operation =
self.ipc_client.subscribe_to_configuration_update(
                key_path=[CONFIGURATION_CLIENT_DEVICE_NAMES],
on_stream_event=self.on_client_device_names_configuration_update_event)
            print(
                'Successfully subscribed to configuration updates for smart
light device names')

    def close_client_device_names_configuration_subscription(self):
        if self.client_device_names_configuration_subscription_operation is not
None:

self.client_device_names_configuration_subscription_operation.close()

    def on_client_device_names_configuration_update_event(self, event):
        try:
            if CONFIGURATION_CLIENT_DEVICE_NAMES in
event.configuration_update_event.key_path:
                print('Received configuration update for list of client
devices')
                self.update_smart_light_device_list()
        except:
            traceback.print_exc()


def choose_random_color():
    return random.choice(COLORS)
```

```
def main():
    try:
        # Create an IPC client and a smart light device manager.
        ipc_client = GreengrassCoreIPCClientV2()
        smart_light_manager = SmartLightDeviceManager(ipc_client)
        smart_light_manager.subscribe_to_shadow_update_accepted_events()

 smart_light_manager.subscribe_to_client_device_name_configuration_updates()
        try:
            # Keep the main thread alive, or the process will exit.
            while True:
                # Set each smart light device to a random color at a regular
 interval.
                for device_name in smart_light_manager.devices:
                    device = smart_light_manager.devices[device_name]
                    desired_color = choose_random_color()
                    print('Chose random color (%s) for %s' %
                          (desired_color, device_name))
                    if desired_color == device.desired_color:
                        print('Desired color for %s is already %s' %
                              (device_name, desired_color))
                    elif desired_color == device.reported_color:
                        print('Reported color for %s is already %s' %
                              (device_name, desired_color))
                    else:
                        smart_light_manager.request_device_color_change(
                            device, desired_color)
                        print('Requested color change for %s to %s' %
                              (device_name, desired_color))
                time.sleep(SET_COLOR_INTERVAL)
        except InterruptedError:
            print('Application interrupted')
        smart_light_manager.close_shadow_update_accepted_subscription()

 smart_light_manager.close_client_device_names_configuration_subscription()
    except Exception:
        print('Exception occurred', file=sys.stderr)
        traceback.print_exc()
        exit(1)


if __name__ == '__main__':
```

```
    main()
```

この Python アプリケーションは次を実行します。

- コンポーネントの設定を読み取って、管理するスマートライトクライアントデバイスのリストを取得します。
- SubscribeToConfigurationUpdate IPC オペレーションを使用して、設定の更新通知にサブスクライブします AWS IoT Greengrass Core ソフトウェアは、コンポーネントの設定が変更されるたびに通知を送信します。コンポーネントは、設定の更新通知を受信すると、管理しているスマートライトでのクライアントデバイスのリストを更新します。
- 各スマートライトの、クライアントデバイスのシャドウを取得して、初期のカラー状態を取得します。
- 各スマートライトのクライアントデバイスの色を 15 秒ごとにランダムに変更します。コンポーネントでは、色を変更するために、クライアントデバイスのモノのシャドウを更新してします。この操作は、MQTT 経由で、クライアントデバイスにシャドウの差分イベントを送信します。
- SubscribeToTopic IPC オペレーションを使用して、ローカルの公開/サブスクライブインターフェイスでの、シャドウアップデート承認済みメッセージにサブスクライブします このコンポーネントは、各スマートライトのクライアントデバイスの色を追跡するために、これらのメッセージを受信します。スマートライトのクライアントデバイスがシャドウの更新を受信すると、MQTT メッセージを送信して、その更新の受信を確認します。MQTT ブリッジは、このメッセージをローカルの公開/サブスクライブインターフェイスに伝達します。

d. Greengrass CLI を使用してコンポーネントをデプロイします。このコンポーネントをデプロイする際は、クライアントデバイス、smartLightDeviceNames、シャドウを管理する対象のリストを指定します。*MyClientDevice1* をクライアントデバイスのモノの名前に置き換えます。

Linux or Unix

```
sudo /greengrass/v2/bin/greengrass-cli deployment create \
  --recipeDir recipes \
  --artifactDir artifacts \
  --merge "com.example.clientdevices.MySmartLightManager=1.0.0" \
  --update-config '{
    "com.example.clientdevices.MySmartLightManager": {
      "MERGE": {
```

```
        "smartLightDeviceNames": [
          "MyClientDevice1"
        ]
      }
    }
  }'
```

Windows Command Prompt (CMD)

```
C:\greengrass\v2/bin/greengrass-cli deployment create ^
  --recipeDir recipes ^
  --artifactDir artifacts ^
  --merge "com.example.clientdevices.MySmartLightManager=1.0.0" ^
  --update-config '{"com.example.clientdevices.MySmartLightManager":
{"MERGE":{"smartLightDeviceNames":["MyClientDevice1"]}}}'
```

PowerShell

```
C:\greengrass\v2/bin/greengrass-cli deployment create `
  --recipeDir recipes `
  --artifactDir artifacts `
  --merge "com.example.clientdevices.MySmartLightManager=1.0.0" `
  --update-config '{
    "com.example.clientdevices.MySmartLightManager": {
      "MERGE": {
        "smartLightDeviceNames": [
          "MyClientDevice1"
        ]
      }
    }
  }'
```

3. コンポーネントログを表示して、コンポーネントのインストールと実行が正常に行われていることを確認します。

   Linux or Unix

   ```
   sudo tail -f /greengrass/v2/logs/
   com.example.clientdevices.MySmartLightManager.log
   ```

PowerShell

```
gc C:\greengrass\v2/logs/com.example.clientdevices.MySmartLightManager.log -Tail
 10 -Wait
```

このコンポーネントは、スマートライトのクライアントデバイスの色を変更するリクエストを送信します。シャドウマネージャーがこのリクエストを受け取り、シャドウの desired ステートをセットします。しかし、スマートライトのクライアントデバイスはまだ動作していないので、シャドウの reported 状態は変更されません。このコンポーネントのログには、次のメッセージが含まれています。

```
2022-07-07T03:49:24.908Z [INFO] (Copier)
 com.example.clientdevices.MySmartLightManager: stdout. Chose random color (blue)
 for MyClientDevice1.
 {scriptName=services.com.example.clientdevices.MySmartLightManager.lifecycle.Run,
 serviceName=com.example.clientdevices.MySmartLightManager, currentState=RUNNING}
2022-07-07T03:49:24.912Z [INFO] (Copier)
 com.example.clientdevices.MySmartLightManager: stdout.
 Requested color change for MyClientDevice1 to blue.
 {scriptName=services.com.example.clientdevices.MySmartLightManager.lifecycle.Run,
 serviceName=com.example.clientdevices.MySmartLightManager, currentState=RUNNING}
```

コンポーネントがメッセージを出力した時点で確認ができるように、このログフィードは開いたままにしておくことができます。

4. Greengrass Discovery を使用しデバイスシャドウの更新をサブスクライブする、サンプルアプリケーションをダウンロードして実行します。クライアントデバイスで、次の操作を行います。

   a. AWS IoT Device SDK v2 for Python のサンプルフォルダに移動します。このサンプルアプリケーションは、サンプルフォルダーにあるコマンドライン解析モジュールを使用します。

   ```
   cd aws-iot-device-sdk-python-v2/samples
   ```

   b. テキストエディタを使用して、次の内容を含む Python スクリプトを、basic_discovery_shadow.py という名前で作成します。このアプリケーションは、Greengrass Discovery とシャドウを使用して、クライアントデバイスとコアデバイス間でのプロパティの同期を維持します。

例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

```
nano basic_discovery_shadow.py
```

ファイルに次の Python コードをコピーします。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0.

from awscrt import io
from awscrt import mqtt
from awsiot import iotshadow
from awsiot.greengrass_discovery import DiscoveryClient
from awsiot import mqtt_connection_builder
from concurrent.futures import Future
import sys
import threading
import traceback
from uuid import uuid4

# Parse arguments
import utils.command_line_utils;
cmdUtils = utils.command_line_utils.CommandLineUtils("Basic Discovery -
 Greengrass discovery example with device shadows.")
cmdUtils.add_common_mqtt_commands()
cmdUtils.add_common_topic_message_commands()
cmdUtils.add_common_logging_commands()
cmdUtils.register_command("key", "<path>", "Path to your key in PEM format.",
 True, str)
cmdUtils.register_command("cert", "<path>", "Path to your client certificate in
 PEM format.", True, str)
cmdUtils.remove_command("endpoint")
cmdUtils.register_command("thing_name", "<str>", "The name assigned to your IoT
 Thing", required=True)
cmdUtils.register_command("region", "<str>", "The region to connect through.",
 required=True)
cmdUtils.register_command("shadow_property", "<str>", "The name of the shadow
 property you want to change (optional, default='color'", default="color")
# Needs to be called so the command utils parse the commands
cmdUtils.get_args()
```

```
# Using globals to simplify sample code
is_sample_done = threading.Event()
mqtt_connection = None
shadow_thing_name = cmdUtils.get_command_required("thing_name")
shadow_property = cmdUtils.get_command("shadow_property")

SHADOW_VALUE_DEFAULT = "off"

class LockedData:
    def __init__(self):
        self.lock = threading.Lock()
        self.shadow_value = None
        self.disconnect_called = False
        self.request_tokens = set()

locked_data = LockedData()


def on_connection_interupted(connection, error, **kwargs):
    print('connection interrupted with error {}'.format(error))


def on_connection_resumed(connection, return_code, session_present, **kwargs):
    print('connection resumed with return code {}, session present
 {}'.format(return_code, session_present))

# Try IoT endpoints until we find one that works
def try_iot_endpoints():
    for gg_group in discover_response.gg_groups:
        for gg_core in gg_group.cores:
            for connectivity_info in gg_core.connectivity:
                try:
                    print('Trying core {} at host {} port
 {}'.format(gg_core.thing_arn, connectivity_info.host_address,
 connectivity_info.port))
                    mqtt_connection = mqtt_connection_builder.mtls_from_path(
                        endpoint=connectivity_info.host_address,
                        port=connectivity_info.port,
                        cert_filepath=cmdUtils.get_command_required("cert"),
                        pri_key_filepath=cmdUtils.get_command_required("key"),

 ca_bytes=gg_group.certificate_authorities[0].encode('utf-8'),
                        on_connection_interrupted=on_connection_interupted,
                        on_connection_resumed=on_connection_resumed,
```

```
                        client_id=cmdUtils.get_command_required("thing_name"),
                        clean_session=False,
                        keep_alive_secs=30)

                    connect_future = mqtt_connection.connect()
                    connect_future.result()
                    print('Connected!')
                    return mqtt_connection

                except Exception as e:
                    print('Connection failed with exception {}'.format(e))
                    continue

    exit('All connection attempts failed')

# Function for gracefully quitting this sample
def exit(msg_or_exception):
    if isinstance(msg_or_exception, Exception):
        print("Exiting sample due to exception.")
        traceback.print_exception(msg_or_exception.__class__, msg_or_exception,
 sys.exc_info()[2])
    else:
        print("Exiting sample:", msg_or_exception)

    with locked_data.lock:
        if not locked_data.disconnect_called:
            print("Disconnecting...")
            locked_data.disconnect_called = True
            future = mqtt_connection.disconnect()
            future.add_done_callback(on_disconnected)

def on_disconnected(disconnect_future):
    # type: (Future) -> None
    print("Disconnected.")

    # Signal that sample is finished
    is_sample_done.set()

def on_get_shadow_accepted(response):
    # type: (iotshadow.GetShadowResponse) -> None
    try:
        with locked_data.lock:
            # check that this is a response to a request from this session
            try:
```

```
                locked_data.request_tokens.remove(response.client_token)
            except KeyError:
                return

            print("Finished getting initial shadow state.")
            if locked_data.shadow_value is not None:
                print("  Ignoring initial query because a delta event has
 already been received.")
                return

        if response.state:
            if response.state.delta:
                value = response.state.delta.get(shadow_property)
                if value:
                    print("  Shadow contains delta value '{}'.".format(value))
                    change_shadow_value(value)
                    return

            if response.state.reported:
                value = response.state.reported.get(shadow_property)
                if value:
                    print("  Shadow contains reported value
 '{}'.".format(value))

 set_local_value_due_to_initial_query(response.state.reported[shadow_property])
                    return

        print("  Shadow document lacks '{}' property. Setting
 defaults...".format(shadow_property))
        change_shadow_value(SHADOW_VALUE_DEFAULT)
        return

    except Exception as e:
        exit(e)

def on_get_shadow_rejected(error):
    # type: (iotshadow.ErrorResponse) -> None
    try:
        # check that this is a response to a request from this session
        with locked_data.lock:
            try:
                locked_data.request_tokens.remove(error.client_token)
            except KeyError:
                return
```

```
        if error.code == 404:
            print("Thing has no shadow document. Creating with defaults...")
            change_shadow_value(SHADOW_VALUE_DEFAULT)
        else:
            exit("Get request was rejected. code:{} message:'{}'".format(
                error.code, error.message))

    except Exception as e:
        exit(e)

def on_shadow_delta_updated(delta):
    # type: (iotshadow.ShadowDeltaUpdatedEvent) -> None
    try:
        print("Received shadow delta event.")
        if delta.state and (shadow_property in delta.state):
            value = delta.state[shadow_property]
            if value is None:
                print("  Delta reports that '{}' was deleted. Resetting
 defaults...".format(shadow_property))
                change_shadow_value(SHADOW_VALUE_DEFAULT)
                return
            else:
                print("  Delta reports that desired value is '{}'. Changing
 local value...".format(value))
                if (delta.client_token is not None):
                    print ("  ClientToken is: " + delta.client_token)
                change_shadow_value(value, delta.client_token)
        else:
            print("  Delta did not report a change in
 '{}'".format(shadow_property))

    except Exception as e:
        exit(e)

def on_publish_update_shadow(future):
    #type: (Future) -> None
    try:
        future.result()
        print("Update request published.")
    except Exception as e:
        print("Failed to publish update request.")
        exit(e)
```

```
def on_update_shadow_accepted(response):
    # type: (iotshadow.UpdateShadowResponse) -> None
    try:
        # check that this is a response to a request from this session
        with locked_data.lock:
            try:
                locked_data.request_tokens.remove(response.client_token)
            except KeyError:
                return

        try:
            if response.state.reported != None:
                if shadow_property in response.state.reported:
                    print("Finished updating reported shadow value to
 '{}'.".format(response.state.reported[shadow_property])) # type: ignore
                else:
                    print ("Could not find shadow property with name:
 '{}'.".format(shadow_property)) # type: ignore
            else:
                print("Shadow states cleared.") # when the shadow states are
 cleared, reported and desired are set to None
        except:
            exit("Updated shadow is missing the target property")

    except Exception as e:
        exit(e)

def on_update_shadow_rejected(error):
    # type: (iotshadow.ErrorResponse) -> None
    try:
        # check that this is a response to a request from this session
        with locked_data.lock:
            try:
                locked_data.request_tokens.remove(error.client_token)
            except KeyError:
                return

        exit("Update request was rejected. code:{} message:'{}'".format(
            error.code, error.message))

    except Exception as e:
        exit(e)

def set_local_value_due_to_initial_query(reported_value):
```

```
    with locked_data.lock:
        locked_data.shadow_value = reported_value

def change_shadow_value(value, token=None):
    with locked_data.lock:
        if locked_data.shadow_value == value:
            print("Local value is already '{}'.".format(value))
            return

        print("Changed local shadow value to '{}'.".format(value))
        locked_data.shadow_value = value

        print("Updating reported shadow value to '{}'...".format(value))

        reuse_token = token is not None
        # use a unique token so we can correlate this "request" message to
        # any "response" messages received on the /accepted and /rejected
 topics
        if not reuse_token:
            token = str(uuid4())

        # if the value is "clear shadow" then send a UpdateShadowRequest with
 None
        # for both reported and desired to clear the shadow document
 completely.
        if value == "clear_shadow":
            tmp_state = iotshadow.ShadowState(reported=None, desired=None,
 reported_is_nullable=True, desired_is_nullable=True)
            request = iotshadow.UpdateShadowRequest(
                thing_name=shadow_thing_name,
                state=tmp_state,
                client_token=token,
            )
        # Otherwise, send a normal update request
        else:
            # if the value is "none" then set it to a Python none object to
            # clear the individual shadow property
            if value == "none":
                value = None

            request = iotshadow.UpdateShadowRequest(
            thing_name=shadow_thing_name,
            state=iotshadow.ShadowState(
                reported={ shadow_property: value }
```

```
                ),
                client_token=token,
            )

        future = shadow_client.publish_update_shadow(request,
 mqtt.QoS.AT_LEAST_ONCE)

        if not reuse_token:
            locked_data.request_tokens.add(token)

        future.add_done_callback(on_publish_update_shadow)


if __name__ == '__main__':
    tls_options =
 io.TlsContextOptions.create_client_with_mtls_from_path(cmdUtils.get_command_required("
 cmdUtils.get_command_required("key"))
    if cmdUtils.get_command(cmdUtils.m_cmd_ca_file):
        tls_options.override_default_trust_store_from_path(None,
 cmdUtils.get_command(cmdUtils.m_cmd_ca_file))
    tls_context = io.ClientTlsContext(tls_options)

    socket_options = io.SocketOptions()

    print('Performing greengrass discovery...')
    discovery_client =
 DiscoveryClient(io.ClientBootstrap.get_or_create_static_default(),
 socket_options, tls_context, cmdUtils.get_command_required("region"))
    resp_future =
 discovery_client.discover(cmdUtils.get_command_required("thing_name"))
    discover_response = resp_future.result()

    print(discover_response)
    if cmdUtils.get_command("print_discover_resp_only"):
        exit(0)

    mqtt_connection = try_iot_endpoints()
    shadow_client = iotshadow.IotShadowClient(mqtt_connection)

    try:
        # Subscribe to necessary topics.
        # Note that is **is** important to wait for "accepted/rejected"
 subscriptions
        # to succeed before publishing the corresponding "request".
```

```
        print("Subscribing to Update responses...")
        update_accepted_subscribed_future, _ =
shadow_client.subscribe_to_update_shadow_accepted(

request=iotshadow.UpdateShadowSubscriptionRequest(thing_name=shadow_thing_name),
            qos=mqtt.QoS.AT_LEAST_ONCE,
            callback=on_update_shadow_accepted)

        update_rejected_subscribed_future, _ =
shadow_client.subscribe_to_update_shadow_rejected(

request=iotshadow.UpdateShadowSubscriptionRequest(thing_name=shadow_thing_name),
            qos=mqtt.QoS.AT_LEAST_ONCE,
            callback=on_update_shadow_rejected)

        # Wait for subscriptions to succeed
        update_accepted_subscribed_future.result()
        update_rejected_subscribed_future.result()

        print("Subscribing to Get responses...")
        get_accepted_subscribed_future, _ =
shadow_client.subscribe_to_get_shadow_accepted(

request=iotshadow.GetShadowSubscriptionRequest(thing_name=shadow_thing_name),
            qos=mqtt.QoS.AT_LEAST_ONCE,
            callback=on_get_shadow_accepted)

        get_rejected_subscribed_future, _ =
shadow_client.subscribe_to_get_shadow_rejected(

request=iotshadow.GetShadowSubscriptionRequest(thing_name=shadow_thing_name),
            qos=mqtt.QoS.AT_LEAST_ONCE,
            callback=on_get_shadow_rejected)

        # Wait for subscriptions to succeed
        get_accepted_subscribed_future.result()
        get_rejected_subscribed_future.result()

        print("Subscribing to Delta events...")
        delta_subscribed_future, _ =
shadow_client.subscribe_to_shadow_delta_updated_events(

request=iotshadow.ShadowDeltaUpdatedSubscriptionRequest(thing_name=shadow_thing_name),
            qos=mqtt.QoS.AT_LEAST_ONCE,
```

```
            callback=on_shadow_delta_updated)

        # Wait for subscription to succeed
        delta_subscribed_future.result()

        # The rest of the sample runs asynchronously.

        # Issue request for shadow's current state.
        # The response will be received by the on_get_accepted() callback
        print("Requesting current shadow state...")

        with locked_data.lock:
            # use a unique token so we can correlate this "request" message to
            # any "response" messages received on the /accepted and /rejected
topics
            token = str(uuid4())

            publish_get_future = shadow_client.publish_get_shadow(

request=iotshadow.GetShadowRequest(thing_name=shadow_thing_name,
client_token=token),
                qos=mqtt.QoS.AT_LEAST_ONCE)

            locked_data.request_tokens.add(token)

        # Ensure that publish succeeds
        publish_get_future.result()

    except Exception as e:
        exit(e)

    # Wait for the sample to finish (user types 'quit', or an error occurs)
    is_sample_done.wait()
```

この Python アプリケーションは次を実行します。

- Greengrass Discovery を使用して、コアデバイスを検出し、それに接続します。
- プロパティの初期状態を取得するために、コアデバイスに対しシャドウドキュメントをリクエストします。
- シャドウの差分イベントにサブスクライブします。このイベントは、プロパティの `desired` 値が `reported` 値と異なった場合に、コアデバイスにより送信されま

す。シャドウの差分イベントを受信したアプリケーションは、プロパティの値を変更し、reported 値に新しい値を設定するためにコアデバイスに更新を送信します。

このアプリケーションでは、AWS IoT Device SDK v2 の Greengrass Discovery とシャドウのサンプルを組み合わせて使用しています。

c. サンプルアプリケーションを実行します。このアプリケーションでは、クライアントデバイスのモノの名前、使用するシャドウプロパティ、および接続の認証および保護のための証明書を、引数で指定する必要があります。

- *MyClientDevice1* をクライアントデバイスのモノの名前に置き換えます。
- *~/certs/AmazonRootCA1.pem* をクライアントデバイスの Amazon ルート CA 証明書へのパスに置き換えます。
- *~/certs/device.pem.crt* をクライアントデバイスのデバイス証明書へのパスに置き換えます。
- *~/certs/private.pem.key* をクライアントデバイスのプライベートキーファイルへのパスに置き換えます。
- *us-east-1* をクライアントデバイスとコアデバイスが動作する AWS リージョンに置き換えます。

```
python3 basic_discovery_shadow.py \
  --thing_name MyClientDevice1 \
  --shadow_property color \
  --ca_file ~/certs/AmazonRootCA1.pem \
  --cert ~/certs/device.pem.crt \
  --key ~/certs/private.pem.key \
  --region us-east-1 \
  --verbosity Warn
```

このサンプルアプリケーションは、シャドウトピックをサブスクライブし、コアデバイスからシャドウの差分イベントを受信するまで待機します。アプリケーションがシャドウの差分イベントを受信して応答したことが、出力に表示されている場合は、クライアントデバイスがコアデバイスのシャドウと正常に対話をできています。

```
Performing greengrass discovery...
awsiot.greengrass_discovery.DiscoverResponse(gg_groups=[awsiot.greengrass_discovery.GGG
coreDevice-MyGreengrassCore',
```

```
 cores=[awsiot.greengrass_discovery.GGCore(thing_arn='arn:aws:iot:us-
east-1:123456789012:thing/MyGreengrassCore',
 connectivity=[awsiot.greengrass_discovery.ConnectivityInfo(id='203.0.113.0',
 host_address='203.0.113.0', metadata='', port=8883)])],
 certificate_authorities=['-----BEGIN CERTIFICATE-----
\nMIICiT...EXAMPLE=\n-----END CERTIFICATE-----\n'])])
Trying core arn:aws:iot:us-east-1:123456789012:thing/MyGreengrassCore at host
 203.0.113.0 port 8883
Connected!
Subscribing to Update responses...
Subscribing to Get responses...
Subscribing to Delta events...
Requesting current shadow state...
Received shadow delta event.
  Delta reports that desired value is 'purple'. Changing local value...
  ClientToken is: 3dce4d3f-e336-41ac-aa4f-7882725f0033
Changed local shadow value to 'purple'.
Updating reported shadow value to 'purple'...
Update request published.
```

アプリケーションが代わりにエラーを出力する場合は、「Greengrass 検出で生じる問題のトラブルシューティング」を参照してください。

コアデバイスの Greengrass ログを表示して、クライアントデバイスが正常に接続してメッセージを送信しているかどうかを確認することもできます。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

5. コンポーネントログをもう一度表示して、スマートライトのクライアントデバイスからのシャドウ更新確認を、そのコンポーネントが受信していることを検証します。

   Linux or Unix

   ```
   sudo tail -f /greengrass/v2/logs/
   com.example.clientdevices.MySmartLightManager.log
   ```

   PowerShell

   ```
   gc C:\greengrass\v2/logs/com.example.clientdevices.MySmartLightManager.log -Tail
    10 -Wait
   ```

コンポーネントは、スマートライトのクライアントデバイスが色を変更したことを、確認するメッセージをログに記録します。

```
2022-07-07T03:49:24.908Z [INFO] (Copier)
 com.example.clientdevices.MySmartLightManager: stdout. Chose random color (blue)
 for MyClientDevice1.
 {scriptName=services.com.example.clientdevices.MySmartLightManager.lifecycle.Run,
 serviceName=com.example.clientdevices.MySmartLightManager, currentState=RUNNING}
2022-07-07T03:49:24.912Z [INFO] (Copier)
 com.example.clientdevices.MySmartLightManager: stdout.
 Requested color change for MyClientDevice1 to blue.
 {scriptName=services.com.example.clientdevices.MySmartLightManager.lifecycle.Run,
 serviceName=com.example.clientdevices.MySmartLightManager, currentState=RUNNING}
2022-07-07T03:49:24.959Z [INFO] (Copier)
 com.example.clientdevices.MySmartLightManager: stdout. Received
 shadow update confirmation from client device: MyClientDevice1.
 {scriptName=services.com.example.clientdevices.MySmartLightManager.lifecycle.Run,
 serviceName=com.example.clientdevices.MySmartLightManager, currentState=RUNNING}
```

> **Note**
>
> クライアントデバイスのシャドウは、コアデバイスとクライアントデバイスの間で同期を維持します。ただしコアデバイスは、AWS IoT Core を使用しているクライアントデバイスのシャドウとは同期しません。フリート内のすべてのデバイスの状態を表示または変更するためには、このシャドウを AWS IoT Core と同期させる必要が生じます。シャドウが AWS IoT Core と同期するようにシャドウマネージャーコンポーネントを設定する方法については、「ローカルデバイスシャドウを AWS IoT Core と同期する」を参照してください。

これで、このチュートリアルは終了です。クライアントデバイスはコアデバイスに接続され、MQTT メッセージを AWS IoT Core と Greengrass コンポーネントに送信し、コアデバイスからシャドウ更新を受信しています。このチュートリアルで説明しているトピックの詳細については、以下を参照してください。

- クライアントデバイスを関連付ける
- コアデバイスのエンドポイントを管理
- クライアントデバイス通信をテストする

- Greengrass Discovery RESTful API
- クライアントデバイスと AWS IoT Core の間の MQTT メッセージのリレー
- コンポーネント内のクライアントデバイスとやり取りする
- デバイスシャドウとやり取り
- クライアントデバイスシャドウとやり取りして同期する

チュートリアル: SageMaker Edge Manager の開始方法

> **Important**
>
> SageMaker Edge Manager は 2024 年 4 月 26 日に廃止されます。エッジデバイスにモデルをデプロイし続ける方法の詳細については、SageMaker 「 Edge Manager のサポート終了」を参照してください。

Amazon SageMaker Edge Manager は、エッジデバイスで実行されるソフトウェアエージェントです。 SageMaker Edge Manager はエッジデバイスのモデル管理を提供するため、Amazon SageMaker Neo でコンパイルされたモデルを Greengrass コアデバイス上で直接パッケージ化して使用できます。 SageMaker Edge Manager を使用すると、コアデバイスからモデルの入力および出力データをサンプリングし、そのデータをモニタリングと分析AWS クラウドのために に送信することもできます。 SageMaker Edge Manager が Greengrass コアデバイスで動作する方法の詳細については、「」を参照してくださいGreengrass コアデバイスで Amazon SageMaker Edge Manager を使用する。

このチュートリアルでは、既存のコアデバイスで AWSが提供するサンプルコンポーネントを使用して SageMaker Edge Manager の使用を開始する方法を示します。これらのサンプルコンポーネントは、 SageMaker Edge Manager コンポーネントを依存関係として使用して Edge Manager エージェントをデプロイし、 SageMaker Neo を使用してコンパイルされた事前トレーニング済みモデルを使用して推論を実行します。 SageMaker Edge Manager エージェントの詳細については、「Amazon SageMaker デベロッパーガイド」の「SageMaker Edge Manager」を参照してください。

既存の Greengrass コアデバイスで SageMaker Edge Manager エージェントをセットアップして使用するために、 は、次のサンプル推論コンポーネントとモデルコンポーネントを作成するために使用できるサンプルコードAWSを提供します。

- イメージ分類

  - `com.greengrass.SageMakerEdgeManager.ImageClassification`
  - `com.greengrass.SageMakerEdgeManager.ImageClassification.Model`
- オブジェクトの検出
  - `com.greengrass.SageMakerEdgeManager.ObjectDetection`
  - `com.greengrass.SageMakerEdgeManager.ObjectDetection.Model`

このチュートリアルでは、サンプルコンポーネントと SageMaker Edge Manager エージェントをデプロイする方法を示します。

トピック

前提条件

このチュートリアルを完了するためには、以下のものが必要です。

- Amazon Linux 2 で実行されている Greengrass コアデバイス、Debian ベースの Linux プラットフォーム (x86_64 またはArmv8)、または Windows (x86_64)。アカウントをお持ちでない場合は、「チュートリアル: AWS IoT Greengrass V2 の開始方法」を参照してください。
- Python 3.6 以降 (ご使用の Python のバージョン用 `pip` がコアデバイスにインストールされていること)。
- コアデバイスにインストールされている OpenGL API GLX ランタイム (`libgl1-mesa-glx`) 。
- 管理者権限を持つ AWS Identity and Access Management (IAM) ユーザー。
- 以下の要件を満たす、インターネットに対応した Windows、Mac、または Unix に類似した開発コンピュータ。
  - Python 3.6 以降がインストールされていること。
  - IAM 管理者ユーザーの認証情報で、AWS CLI がインストールおよび設定されていること。詳細については、「AWS CLI のインストール」と「AWS CLI の設定」を参照してください。
- Greengrass コアデバイスと同じ AWS アカウント と AWS リージョン で作成された次の S3 バケット。

- サンプル推論およびモデルコンポーネントに含まれるアーティファクトを格納する S3 バケット。このチュートリアルでは、このバケットを参照するにあたり *DOC-EXAMPLE-BUCKET1* を使用します。
- SageMaker エッジデバイスフリートに関連付ける S3 バケット。 SageMaker Edge Manager には、エッジデバイスフリートを作成し、デバイスで実行中の推論からサンプルデータを保存するための S3 バケットが必要です。このチュートリアルでは、このバケットを参照するにあたり *DOC-EXAMPLE-BUCKET2* を使用します。

S3 バケットを作成する方法の情報については、「Amazon S3 の使用を開始」を参照してください。

- 次のように設定された Greengrass デバイスのロール:
  - 次の IAM ポリシーの例で示されているように、`credentials.iot.amazonaws.com` と `sagemaker.amazonaws.com` がロールの継承を可能にする信頼関係。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.iot.amazonaws.com"
       },
      "Action": "sts:AssumeRole"
    },
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "sagemaker.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

  - IAM AmazonSageMakerEdgeDeviceFleetPolicy マネージドポリシー。
  - IAM AmazonSageMakerFullAccess 管理ポリシー。
  - 次の IAM ポリシー例に示されているように、コンポーネントのアーティファクトが含まれる S3 バケットに対する `s3:GetObject` アクション。

```
{
  "Version": "2012-10-17",
  "Statement": [
  {
    "Action": [
      "s3:GetObject"
    ],
    "Resource": [
      "arn:aws:s3:::DOC-EXAMPLE-BUCKET1/*"
    ],
    "Effect": "Allow"
  }
  ]
}
```

SageMaker Edge Manager で Greengrass コアデバイスをセットアップする

SageMaker Edge Manager のエッジデバイスフリートは、論理的にグループ化されたデバイスのコレクションです。で SageMaker Edge Manager を使用するにはAWS IoT Greengrass、Edge Manager エージェントをデプロイする Greengrass コアデバイスと同じAWS IoTロールエイリアスを使用する SageMaker エッジデバイスフリートを作成する必要があります。その後、コアデバイスをそのフリートの一部として登録する必要があります。

トピック

- エッジデバイスフリートを作成する
- Greengrass コアデバイスを登録する

エッジデバイスフリートを作成する

エッジデバイスフリートを作成するには (コンソール)

1. Amazon SageMaker コンソール で、エッジマネージャー を選択し、エッジデバイスフリートを選択します。
2. [Device fleets] (デバイスフリート) ページで、[Create device fleet] (デバイスフリートを作成) を選択します。

3. [Device fleet properties] (デバイスフリートのプロパティ) で、次の作業を行います。
   - [Device fleet name] (デバイスフリート名) には、デバイスフリートの名前を入力します。
   - [IAM role] (IAM ロール) には、Greengrass コアデバイスをセットアップするときに指定した AWS IoT ロールエイリアスの Amazon リソースネーム (ARN) を入力します。
   - [Create IAM role alias] (IAM ロールエイリアスを作成) のトグルを無効にします。
4. [次へ] をクリックします。
5. [Output configuration] (出力の設定) で、[S3 bucket URI] (S3 バケット URI) にデバイスフリートに関連付ける S3 バケットの URI を入力します。
6. [送信] を選択します。

Greengrass コアデバイスを登録する

Greengrass コアデバイスをエッジデバイスとして登録するには (コンソール)

1. Amazon SageMaker コンソール で、エッジマネージャー を選択し、エッジデバイス を選択します。
2. [Devices] (デバイス) ページで、[Register devices] (デバイスの登録) を選択します。
3. [Device properties] (デバイスのプロパティ) では、[Device fleet name] (デバイスフリート名) に作成したデバイスフリートの名前を入力し、[Next] (次へ) を選択します。
4. [次へ] をクリックします。
5. [Device source] (デバイスソース) では、[Device name] (デバイス名) に Greengrass コアデバイスの AWS IoT モノ名を入力します。
6. [送信] を選択します。

サンプルコンポーネントを作成する

SageMaker Edge Manager コンポーネントの使用を開始する際に役立つように、 AWSには、サンプル推論とモデルコンポーネントを作成し、 にアップロードする Python スクリプト GitHub が用意されていますAWS クラウド。開発用コンピュータで以下の手順を完了します。

サンプルコンポーネントを作成するには

1. のAWS IoT Greengrassコンポーネントサンプルリポジトリを開発コンピュータ GitHub にダウンロードします。

2. ダウンロードした /machine-learning/sagemaker-edge-manager フォルダに移動します。

```
cd download-directory/machine-learning/sagemaker-edge-manager
```

3. 次のコマンドを実行して、サンプルコンポーネントを作成し、AWS クラウド にアップロードします。

```
python3 create_components.py -r region -b DOC-EXAMPLE-BUCKET
```

region を Greengrass コアデバイスを作成した AWS リージョン に置き換えて、*DOC-EXAMPLE-BUCKET1* をコンポーネントアーティファクトを保存する S3 バケットの名前に置き換えます。

> **Note**
>
> デフォルトでは、スクリプトはイメージ分類推論とオブジェクト検出推論の両方に対するサンプルコンポーネントを作成します。特定のタイプの推論に対してのみのコンポーネントを作成する場合は、`-i ImageClassification | ObjectDetection` 引数を指定します。

SageMaker Edge Manager で使用するサンプル推論コンポーネントとモデルコンポーネントが に作成されるようになりましたAWS アカウント。AWS IoT Greengrass コンソールでサンプルコンポーネントを表示するには、[Components] (コンポーネント) を選択し、[My components] (マイコンポーネント) で次のコンポーネントを検索します。

- `com.greengrass.SageMakerEdgeManager.ImageClassification`
- `com.greengrass.SageMakerEdgeManager.ImageClassification.Model`
- `com.greengrass.SageMakerEdgeManager.ObjectDetection`
- `com.greengrass.SageMakerEdgeManager.ObjectDetection.Model`

サンプルイメージ分類推論を実行する

AWSが提供するサンプルコンポーネントと SageMaker Edge Manager エージェントを使用してイメージ分類推論を実行するには、これらのコンポーネントをコアデバイスにデプロイする必要があります。これらのコンポーネントをデプロイすると、 SageMaker Neo でコンパイル済みの事前トレーニング済み Resnet-50 モデルがダウンロードされ、デバイスに SageMaker Edge Manager エージェ

ントがインストールされます。 SageMaker Edge Manager エージェントはモデルをロードし、推論結果を `gg/sageMakerEdgeManager/image-classification`トピックに発行します。これらの推論結果を確認するには、AWS IoT コンソールの AWS IoT MQTT クライアントを使用してこのトピックをサブスクライブします。

トピック

- 通知トピックをサブスクライブする
- サンプルコンポーネントをデプロイする
- 推論結果を表示する

通知トピックをサブスクライブする

このステップでは、AWS IoT コンソールに AWS IoT MQTT クライアントを設定して、サンプル推論コンポーネントによってパブリッシュされた MQTT メッセージを確認します。デフォルトでは、コンポーネントは推論結果を `gg/sageMakerEdgeManager/image-classification` トピックにパブリッシュします。コンポーネントを Greengrass コアデバイスにデプロイする前に、このトピックにサブスクライブして、コンポーネントが初めて実行されたときの推論結果を確認します。

デフォルトの通知トピックへサブスクライブするには

1. AWS IoT コンソールのナビゲーションメニューで、[Test, MQTT test client] (テスト、MQTT テストクライアント) を選択します。
2. Subscribe to a topic (トピックへサブスクライブ) 内の[Topic name] (トピック名) ボックスで、**`gg/sageMakerEdgeManager/image-classification`** と入力します。
3. [サブスクライブ] を選択します。

サンプルコンポーネントをデプロイする

このステップでは、次のコンポーネントを設定し、コアデバイスにデプロイします。

- `aws.greengrass.SageMakerEdgeManager`
- `com.greengrass.SageMakerEdgeManager.ImageClassification`
- `com.greengrass.SageMakerEdgeManager.ImageClassification.Model`

コンポーネントをデプロイするには (コンソール)

1. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Deployments] (デプロイ) を選択した後、修正するターゲットデバイスのデプロイを選択します。
2. [deployment] (デプロイデプロイ) ページで、[Revise] (修正) を選択し、[Revise deployment] (デプロイの修正) を選択します。
3. [Specify target] (ターゲットの指定) ページで [Next] (次へ) を選択します。
4. [Select components] (コンポーネントの選択) ページで、次の手順を実行します:
   a. [My components] (マイコンポーネント) で、次のコンポーネントを選択します。
      - `com.greengrass.SageMakerEdgeManager.ImageClassification`
      - `com.greengrass.SageMakerEdgeManager.ImageClassification.Model`
   b. [Public components] (パブリックコンポーネント) で、[Show only selected components] (選択したコンポーネントのみを表示) トグルをオフにしてから、`aws.greengrass.SageMakerEdgeManager` コンポーネントを選択します。
   c. [次へ] をクリックします。
5. [Configure components] (コンポーネントを設定) ページで、`aws.greengrass.SageMakerEdgeManager` コンポーネントを選択したら、次の操作を行います。
   a. [Configure component] (コンポーネントを設定) を選択します。
   b. [Configuration update] (設定を更新) の [Configuration to merge] (マージの設定) で、次の設定を入力します。

      ```
      {
          "DeviceFleetName": "device-fleet-name",
          "BucketName": "DOC-EXAMPLE-BUCKET"
      }
      ```

      を作成したエッジデバイスフリートの名前*device-fleet-name*に置き換え、*DOC-EXAMPLE-BUCKET* をデバイスフリートに関連付けられている S3 バケットの名前に置き換えます。
   c. [Confirm] (確認)、[Next] (次へ) の順に選択します。
6. [Configure advanced settings] (詳細設定) ページはデフォルト設定のままにし、[Next] (次へ) を選択します。

7. [Review] (レビュー) ページで、[Deploy] (デプロイ) を選択します。

コンポーネントをデプロイするには (AWS CLI)

1. 開発コンピュータで、 SageMaker ファイルを作成して deployment.json Edge Manager コンポーネントのデプロイ設定を定義します。このファイルは、次の例のようになります。

```
{
  "targetArn":"targetArn",
  "components": {
    "aws.greengrass.SageMakerEdgeManager": {
      "componentVersion": "1.0.x",
      "configurationUpdate": {
        "merge": "{\"DeviceFleetName\":\"device-fleet-name\",\"BucketName\":\"DOC-
EXAMPLE-BUCKET2\"}"
      }
    },
    "com.greengrass.SageMakerEdgeManager.ImageClassification": {
      "componentVersion": "1.0.x",
      "configurationUpdate": {
      }
    },
    "com.greengrass.SageMakerEdgeManager.ImageClassification.Model": {
      "componentVersion": "1.0.x",
      "configurationUpdate": {
      }
    },
  }
}
```

- [targetArn] フィールドで *targetArn* をデプロイメントの対象となるモノまたはモノのグループの Amazon リソースネーム (ARN) に置き換えます。形式は以下のとおりです:
  - モノ: arn:aws:iot:*region*:*account-id*:thing/*thingName*
  - モノのグループ: arn:aws:iot:*region*:*account-id*:thinggroup/*thingGroupName*
- merge フィールドで、 を、作成したエッジデバイスフリートの名前*device-fleet-name*に置き換えます。次に、*DOC-EXAMPLE-BUCKET2* をデバイスフリートに関連付けられた S3 バケットの名前に置き換えます。
- 各コンポーネントのコンポーネントバージョンを、使用可能な最新のバージョンに置き換えます。

2. 次のコマンドを実行して、デバイスにコンポーネントをデプロイします。

```
aws greengrassv2 create-deployment \
    --cli-input-json file://path/to/deployment.json
```

デプロイには数分かかる場合があります。次のステップでは、コンポーネントログをチェックして、デプロイが正常に完了したことを確認し、推論結果を表示します。

推論結果を表示する

コンポーネントをデプロイした後は、Greengrass コアデバイスのコンポーネントログと、AWS IoT コンソールの AWS IoT MQTT クライアントで推論結果を確認できます。コンポーネントが推論結果をパブリッシュするトピックへサブスクライブするには、「通知トピックをサブスクライブする」を参照してください。

- AWS IoT MQTT クライアント - 推論コンポーネントがデフォルトの通知トピックにパブリッシュした結果を確認するには、次のステップを実行します:

  1. AWS IoT コンソールのナビゲーションメニューで、[Test, MQTT test client] (テスト、MQTT テストクライアント) を選択します。
  2. [Subscriptions] (サブスクリプション) 内で、**gg/sageMakerEdgeManager/image-classification** を選択します。

- コンポーネントログ - コンポーネントログで推論結果を確認するには、Greengrass コアデバイスで次のコマンドを実行します。

```
sudo tail -f /greengrass/v2/logs/
com.greengrass.SageMakerEdgeManager.ImageClassification.log
```

コンポーネントログまたは MQTT クライアントで推論結果が確認できない場合、デプロイが失敗しているか、コアデバイスに到達していません。これは、コアデバイスがインターネットに接続されていない、あるいはコンポーネントを実行するために適切な権限がない場合に発生します。コアデバイスで次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアのログファイルを確認します。このファイルは、Greengrass コアデバイスのデプロイサービスからのログが含まれます。

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

詳細については、「機械学習の推論に対するトラブルシューティング」を参照してください。

チュートリアル: TensorFlow Lite を使用してサンプルイメージ分類推論を実行する

このチュートリアルでは、TensorFlow Lite イメージ分類推論コンポーネントを使用して、Greengrass コアデバイスでサンプルイメージ分類推論を実行する方法を示します。このコンポーネントには、次のコンポーネントの従属関係が含まれます:

- TensorFlow Lite イメージ分類モデルストアコンポーネント
- TensorFlow Lite ランタイムコンポーネント

このコンポーネントをデプロイすると、事前トレーニング済みの MobileNet v1 モデルがダウンロードされ、TensorFlow Lite ランタイムとその依存関係がインストールされます。このコンポーネントは推論結果を `ml/tflite/image-classification` トピックにパブリッシュします。これらの推論結果を確認するには、AWS IoT コンソールの AWS IoT MQTT クライアントを使用してこのトピックをサブスクライブします。

このチュートリアルでは、サンプル推論コンポーネントをデプロイして、AWS IoT Greengrass によって提供されるサンプル画像にイメージ分類を実行します。このチュートリアルを完了すると、チュートリアル: TensorFlow Lite を使用してカメラからの画像に対してサンプル画像分類推論を実行する を完了することができます。これには、サンプル推論コンポーネントを修正して、Greengrass コアデバイスのカメラからローカルで画像にイメージ分類を実行する方法を説明します。

Greengrass デバイスでの機械学習の詳細については、「機械学習の推論を実行する」を参照してください。

トピック

前提条件

このチュートリアルを完了するには、以下が必要です。

- Linux Greengrass コアデバイス。アカウントをお持ちでない場合は、「チュートリアル: AWS IoT Greengrass V2 の開始方法」を参照してください。コアデバイスは、次の要件を満たしている必要があります。
  - Amazon Linux 2 または Ubuntu 18.04 を実行している Greengrass コアデバイスの場合は、GNU C ライブラリ (glibc) バージョン 2.27 以降がデバイスにインストールされている必要があります。
  - Raspberry Pi などの Armv7l デバイスでは、OpenCV-Python の依存関係がデバイスにインストールされています。次のコマンドを実行して依存関係をインストールします。

    ```
    sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
     libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
    ```

  - Raspberry Pi OS Bullseye を実行する Raspberry Pi デバイスでは、次の要件を満たす必要があります。
    - NumPy 1.22.4 以降がデバイスにインストールされている。Raspberry Pi OS Bullseye には以前のバージョンの が含まれているため NumPy、次のコマンドを実行して NumPy デバイスでアップグレードできます。

      ```
      pip3 install --upgrade numpy
      ```

    - デバイスで、レガシーカメラスタックが有効になっていること。Raspberry Pi OS Bullseye には、デフォルトで新しいカメラスタックが含まれており有効化されていますが、これには互換性がないるため、レガシーカメラスタックを有効にしておく必要があります。

      レガシーカメラスタックを有効にするには

      1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

         ```
         sudo raspi-config
         ```

      2. [Interface Options] (インターフェイスオプション) を選択します。
      3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
      4. Raspberry Pi を再起動します。

ステップ 1: デフォルトの通知トピックへサブスクライブ

このステップでは、 TensorFlow Lite イメージ分類コンポーネントによって発行された AWS IoT MQTT メッセージを監視するようにAWS IoT、コンソールで MQTT クライアントを設定します。デフォルトでは、コンポーネントは推論結果を `ml/tflite/image-classification` トピックにパブリッシュします。コンポーネントを Greengrass コアデバイスにデプロイする前に、このトピックにサブスクライブして、コンポーネントが初めて実行されたときの推論結果を確認します。

デフォルトの通知トピックへサブスクライブするには

1. AWS IoT コンソールのナビゲーションメニューで、[Test, MQTT test client] (テスト、MQTT テストクライアント) を選択します。
2. Subscribe to a topic (トピックへサブスクライブ) 内の[Topic name] (トピック名) ボックスで、**`ml/tflite/image-classification`** と入力します。
3. [サブスクライブ] を選択します。

ステップ 2: TensorFlow Lite イメージ分類コンポーネントをデプロイする

このステップでは、 TensorFlow Lite イメージ分類コンポーネントをコアデバイスにデプロイします。

TensorFlow Lite イメージ分類コンポーネントをデプロイするには (コンソール)

1. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Components] (コンポーネント) を選択します。
2. [Components] (コンポーネント) ページの [Public components] (公開コンポーネント) タブで、[`aws.greengrass.TensorFlowLiteImageClassification`] を選択します。
3. [aws.greengrass.TensorFlowLiteImageClassification] ページで、[Deploy] (デプロイ) を選択します。
4. [Add to deployment] (デプロイに追加) から、次のいずれかを選択します。
   a. ターゲットデバイスにある既存のデプロイにこのコンポーネントをマージするには、[Add to existing deployment] (既存のデプロイに追加) をクリックし、修正するデプロイを選択します。

b. ターゲットデバイスに新しいデプロイを作成するには、[Create new deployment] (新しいデプロイの作成) を選択します。デバイスに既存のデプロイがある場合は、このステップを選択すると既存のデプロイが置き換えられます。

5. [Specify device state] (ターゲットを指定) ページで、次を実行します。

   a. [Deployment information] (デプロイ情報) で、デプロイの名前を入力または変更して、わかりやすくします。

   b. [Deployment targets] (デプロイターゲット) でデプロイのターゲットを選択し、[Next] (次へ) を選択します。既存のデプロイを修正する場合は、デプロイターゲットを変更できません。

6. [Select components] (コンポーネントを選択) ページの [Public components] (パブリックコンポーネント) 内で、`aws.greengrass.TensorFlowLiteImageClassification` コンポーネントが選択されていることを確認して、[Next] (次) を選択します。

7. [Configure components] (コンポーネント設定) ページで、デフォルト設定のままにして、[Next] (次へ) を選択します。

8. [Configure advanced settings] (詳細設定) ページはデフォルト設定のままにし、[Next] (次へ) を選択します。

9. [Review] (レビュー) ページで、[Deploy] (デプロイ) を選択します。

TensorFlow Lite イメージ分類コンポーネント (AWS CLI) をデプロイするには

1. TensorFlow Lite イメージ分類コンポーネントのデプロイ設定を定義する `deployment.json` ファイルを作成します。このファイルは次のようになります:

```
{
  "targetArn":"targetArn",
  "components": {
    "aws.greengrass.TensorFlowLiteImageClassification": {
      "componentVersion": 2.1.0,
      "configurationUpdate": {
      }
    }
  }
}
```

   - [`targetArn`] フィールドで *`targetArn`* をデプロイメントの対象となるモノまたはモノのグループの Amazon リソースネーム (ARN) に置き換えます。形式は以下のとおりです:
     - モノ: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`thingName`*

- モノのグループ: `arn:aws:iot:region:account-id:thinggroup/thingGroupName`
- このチュートリアルでは、コンポーネントバージョン 2.1.0 を使用します。`aws.greengrass.TensorFlowLiteObjectDetection` コンポーネントオブジェクトで *2.1.0* を置き換えて、別のバージョンの TensorFlow Lite オブジェクト検出コンポーネントを使用します。

2. 次のコマンドを実行して、デバイスに TensorFlow Lite イメージ分類コンポーネントをデプロイします。

```
aws greengrassv2 create-deployment \
    --cli-input-json file://path/to/deployment.json
```

デプロイには数分かかる場合があります。次のステップでは、コンポーネントログをチェックして、デプロイが正常に完了したことを確認し、推論結果を表示します。

ステップ 3: 推論結果の確認

コンポーネントをデプロイした後、Greengrass コアデバイスのコンポーネントログと、AWS IoT コンソールの AWS IoT MQTT クライアントで推論結果を確認できます。コンポーネントが推論結果をパブリッシュするトピックへサブスクライブするには、「ステップ 1: デフォルトの通知トピックへサブスクライブ」を参照してください。

- AWS IoT MQTT クライアント - 推論コンポーネントがデフォルトの通知トピックにパブリッシュした結果を確認するには、次のステップを実行します:

  1. AWS IoT コンソールのナビゲーションメニューで、[Test, MQTT test client] (テスト、MQTT テストクライアント) を選択します。
  2. [Subscriptions] (サブスクリプション) 内で、**ml/tflite/image-classification** を選択します。

     次の例に示すようなメッセージが表示されます。

```
{
  "timestamp": "2021-01-01 00:00:00.000000",
  "inference-type": "image-classification",
  "inference-description": "Top 5 predictions with score 0.3 or above ",
  "inference-results": [
    {
```

```
      "Label": "cougar, puma, catamount, mountain lion, painter, panther, Felis
 concolor",
      "Score": "0.5882352941176471"
    },
    {
      "Label": "Persian cat",
      "Score": "0.5882352941176471"
    },
    {
      "Label": "tiger cat",
      "Score": "0.5882352941176471"
    },
    {
      "Label": "dalmatian, coach dog, carriage dog",
      "Score": "0.5607843137254902"
    },
    {
      "Label": "malamute, malemute, Alaskan malamute",
      "Score": "0.5450980392156862"
    }
  ]
}
```

- コンポーネントログ - コンポーネントログで推論結果を確認するには、Greengrass コアデバイスで次のコマンドを実行します。

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.TensorFlowLiteImageClassification.log
  ```

  次の例のような結果が表示されます。

  ```
  2021-01-01 00:00:00.000000 [INFO] (Copier)
   aws.greengrass.TensorFlowLiteImageClassification: stdout. Publishing results to the
   IoT core....
   {scriptName=services.aws.greengrass.TensorFlowLiteImageClassification.lifecycle.Run.script,
   serviceName=aws.greengrass.TensorFlowLiteImageClassification, currentState=RUNNING}

  2021-01-01 00:00:00.000000 [INFO] (Copier)
   aws.greengrass.TensorFlowLiteImageClassification: stdout. {"timestamp":
   "2021-01-01 00:00:00.000000", "inference-type": "image-classification", "inference-
  description": "Top 5 predictions with score 0.3 or above ", "inference-results":
   [{"Label": "cougar, puma, catamount, mountain lion, painter, panther, Felis
   concolor", "Score": "0.5882352941176471"}, {"Label": "Persian cat", "Score":
   "0.5882352941176471"}, {"Label": "tiger cat", "Score": "0.5882352941176471"},
  ```

```
{"Label": "dalmatian, coach dog, carriage dog", "Score": "0.5607843137254902"},
{"Label": "malamute, malemute, Alaskan malamute", "Score": "0.5450980392156862"}]}.
{scriptName=services.aws.greengrass.TensorFlowLiteImageClassification.lifecycle.Run.script,
serviceName=aws.greengrass.TensorFlowLiteImageClassification, currentState=RUNNING}
```

コンポーネントログまたは MQTT クライアントで推論結果が確認できない場合、デプロイが失敗しているか、コアデバイスに到達していません。これは、コアデバイスがインターネットに接続されていない、あるいはコンポーネントを実行するために適切な権限がない場合に発生します。コアデバイスで次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアのログファイルを確認します。このファイルは、Greengrass コアデバイスのデプロイサービスからのログが含まれます。

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

詳細については、「機械学習の推論に対するトラブルシューティング」を参照してください。

次のステップ

サポートされたカメラインターフェースを備えた Greengrass コアデバイスをお持ちの場合、チュートリアル: TensorFlow Lite を使用してカメラからの画像に対してサンプル画像分類推論を実行するを完了できます。これは、サンプル推論コンポーネントを修正して、カメラの画像にイメージ分類を実行する方法を説明します。

サンプルの TensorFlow Lite イメージ分類推論コンポーネントの設定をさらに調べるには、以下を試してください。

- `InferenceInterval` 設定パラメータを修正して推論コードの実行頻度を変更します。
- 推論コンポーネントコンフィギュレーションの `ImageName` と `ImageDirectory` 設定パラメータを修正して、推論に使用するカスタムイメージを指定します。

パブリックコンポーネントの設定のカスタマイズ、またはカスタム機械学習コンポーネントの作成に関する情報については、「機械学習コンポーネントのカスタマイズ」を参照してください。

チュートリアル: TensorFlow Lite を使用してカメラからの画像に対してサンプル画像分類推論を実行する

このチュートリアルでは、TensorFlow Lite イメージ分類推論コンポーネントを使用して、Greengrass コアデバイスでローカルにカメラから画像に対してサンプルイメージ分類推論を実行する方法を示します。このコンポーネントには、次のコンポーネントの従属関係が含まれます:

- TensorFlow Lite イメージ分類モデルストアコンポーネント
- TensorFlow Lite ランタイムコンポーネント

> **Note**
>
> このチュートリアルでは、Raspberry Pi もしくは NVIDIA Jetson Nano デバイスのカメラモジュールにアクセスしていますが、AWS IoT Greengrass では、Armv7l、Armv8、または x86_64 プラットフォームの他のデバイスもサポートしています。別のデバイスにカメラを設定するには、デバイスの関連マニュアルを参照してください。

Greengrass デバイスでの機械学習の詳細については、「機械学習の推論を実行する」を参照してください。

トピック

前提条件

このチュートリアルを完了するには、まず チュートリアル: TensorFlow Lite を使用してサンプルイメージ分類推論を実行する を完了する必要があります。

また、以下も必要になります:

- カメラインターフェイスを備えた Linux Greengrass コアデバイス。このチュートリアルでは、サポートされている次に示すデバイスのいずれかのカメラモジュールにアクセスします:
  - Raspberry Pi OS (以前はRaspbianと呼ばれていた) を実行中の Raspberry Pi
  - NVIDIA Jetson Nano

  Greengrass コアデバイスのセットアップの情報については、「チュートリアル: AWS IoT Greengrass V2 の開始方法」を参照してください。

  コアデバイスは、次の要件を満たしている必要があります。

  - Amazon Linux 2 または Ubuntu 18.04 を実行している Greengrass コアデバイスの場合は、GNU C ライブラリ (glibc) バージョン 2.27 以降がデバイスにインストールされている必要があります。
  - Raspberry Pi などの Armv7l デバイスでは、OpenCV-Python の依存関係がデバイスにインストールされています。次のコマンドを実行して依存関係をインストールします。

    ```
    sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
     libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
    ```

  - Raspberry Pi OS Bullseye を実行する Raspberry Pi デバイスでは、次の要件を満たす必要があります。
    - NumPy 1.22.4 以降がデバイスにインストールされている。Raspberry Pi OSwireseye には以前のバージョンの が含まれているため NumPy、次のコマンドを実行して NumPy デバイスでアップグレードできます。

      ```
      pip3 install --upgrade numpy
      ```

    - デバイスで、レガシーカメラスタックが有効になっていること。Raspberry Pi OS Bullseye には、デフォルトで新しいカメラスタックが含まれており有効化されていますが、これには互換性がないるため、レガシーカメラスタックを有効にしておく必要があります。

      レガシーカメラスタックを有効にするには

      1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

         ```
         sudo raspi-config
         ```

      2. [Interface Options] (インターフェイスオプション) を選択します。

3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
4. Raspberry Pi を再起動します。

- Raspberry Pi または NVIDIA Jetson Nano デバイスの場合、Raspberry Pi カメラモジュール V2 - 8 megapixel、1080p。カメラの設定方法については、Raspberry Pi ドキュメントで「カメラの接続」を参照してください。

ステップ 1: デバイスのカメラモジュールを設定

このステップでは、デバイスのカメラモジュールをインストールして有効にします。デバイスに次のコマンドを実行します。

Raspberry Pi (Armv7l)

1. カメラモジュールの `picamera` インターフェイスをインストールします。次のコマンドを実行して、カメラモジュールとこのチュートリアルに必要な他の Python ライブラリをインストールします。

```
sudo apt-get install -y python3-picamera
```

2. Picamera が正常にインストールされたことを確認します。

```
sudo -u ggc_user bash -c 'python3 -c "import picamera"'
```

出力にエラーが含まれていない場合、検証は成功です。

> **Note**
>
> デバイスにインストールされた Python 実行可能ファイルが `python3.7` である場合、このチュートリアルのコマンドに `python3` ではなく、`python3.7` を使用します。依存関係のエラーを回避するために、pip インストールが正しい `python3` バージョンまたは `python3.7` バージョンにマップされていることを確認してください。

3. デバイスを再起動します。

```
sudo reboot
```

4. Raspberry Pi 設定ツールを開きます。

```
sudo raspi-config
```

5. 矢印キーを使用して [Interfacing Options] (インターフェイスオプション) を開き、カメラインターフェイスを有効にします。プロンプトが表示されたら、デバイスを再起動します。
6. 次のコマンドを実行してカメラの設定をテストします。

```
raspistill -v -o test.jpg
```

これにより、Raspberry Pi のプレビューウィンドウが開き、test.jpg という写真が現在のディレクトリに保存されて、カメラに関する情報が Raspberry Pi のターミナルに表示されます。

7. 次のコマンドを実行し、シンボリックリンクを作成して、推論コンポーネントがランタイムコンポーネントによって作成されたバーチャル環境からカメラにアクセスできるようにします。

```
sudo ln -s /usr/lib/python3/dist-packages/picamera "MLRootPath/
greengrass_ml_tflite_venv/lib/python3.7/site-packages"
```

このチュートリアルの *MLRootPath* のデフォルト値は です*/greengrass/v2*/work/variant.TensorFlowLite/greengrass_ml。この場所の greengrass_ml_tflite_venv フォルダは、チュートリアル: TensorFlow Lite を使用してサンプルイメージ分類推論を実行する で初めて推論コンポーネントをデプロイするときに作成されます。

Jetson Nano (Armv8)

1. 次のコマンドを実行してカメラの設定をテストします。

```
gst-launch-1.0 nvarguscamerasrc num-buffers=1 ! "video/x-raw(memory:NVMM),
 width=1920, height=1080, format=NV12, framerate=30/1" ! nvjpegenc ! filesink
 location=test.jpg
```

これは test.jpg という名前のイメージをキャプチャして現在のディレクトリに保存します。

2. (オプション) デバイスを再起動します。前の手順で gst-launch コマンドを実行時に問題が発生した場合、デバイスを再起動するとこれらの問題が解決される場合があります。

```
sudo reboot
```

> **Note**
>
> Jetson Nano など、Armv8 (Aarch64) デバイスの場合、推論コンポーネントが、ランタイムコンポーネントによって作成されたバーチャル環境からカメラにアクセスできるようにするシンボリックリンクを作成する必要はありません。

ステップ 2: デフォルトの通知トピックのサブスクライブ状況を確認

でチュートリアル: TensorFlow Lite を使用してサンプルイメージ分類推論を実行する、 AWS IoT `ml/tflite/image-classification`トピックの TensorFlow Lite イメージ分類コンポーネントによって発行された MQTT メッセージを監視するように、 AWS IoTコンソールで MQTT クライアントを設定しました。AWS IoT コンソールで、このサブスクリプションが存在することを確認します。そうでない場合、ステップ 1: デフォルトの通知トピックへサブスクライブ の手順に従って、コンポーネントを Greengrass コアデバイスにデプロイする前に、このトピックをサブスクライブします。

ステップ 3: TensorFlow Lite イメージ分類コンポーネント設定を変更してデプロイする

このステップでは、 TensorFlow Lite イメージ分類コンポーネントを設定し、コアデバイスにデプロイします。

TensorFlow Lite イメージ分類コンポーネントを設定してデプロイするには (コンソール)

1. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Components] (コンポーネント) を選択します。
2. [Components] (コンポーネント) ページの [Public components] (公開コンポーネント) タブで、[`aws.greengrass.TensorFlowLiteImageClassification`] を選択します。
3. [aws.greengrass.TensorFlowLiteImageClassification] ページで、[Deploy] (デプロイ) を選択します。
4. [Add to deployment] (デプロイに追加) から、次のいずれかを選択します。

a. ターゲットデバイスにある既存のデプロイにこのコンポーネントをマージするには、[Add to existing deployment] (既存のデプロイに追加) をクリックし、修正するデプロイを選択します。

b. ターゲットデバイスに新しいデプロイを作成するには、[Create new deployment] (新しいデプロイの作成) を選択します。デバイスに既存のデプロイがある場合は、このステップを選択すると既存のデプロイが置き換えられます。

5. [Specify device state] (ターゲットを指定) ページで、次を実行します。

a. [Deployment information] (デプロイ情報) で、デプロイの名前を入力または変更して、わかりやすくします。

b. [Deployment targets] (デプロイターゲット) でデプロイのターゲットを選択し、[Next] (次へ) を選択します。既存のデプロイを修正する場合は、デプロイターゲットを変更できません。

6. [Select components] (コンポーネントを選択) ページの [Public components] (パブリックコンポーネント) 内で、`aws.greengrass.TensorFlowLiteImageClassification` コンポーネントが選択されていることを確認して、[Next] (次) を選択します。

7. [Configure components] (コンポーネントの設定) ページで、次の手順を実行します:

a. 推論コンポーネントを選択して、[Configure component] (コンポーネントの設定) を選択します。

b. [Configuration update] (設定更新) 内で、[Configuration to merge] (マージの設定) ボックスに次の設定更新を入力します:

```
{
  "InferenceInterval": "60",
  "UseCamera": "true"
}
```

この設定更新により、コンポーネントはデバイスのカメラモジュールにアクセスし、カメラで撮影した画像の推論を実行します。推論コードは 60 秒ごとに実行されます。

c. [Confirm] (確認)、[Next] (次へ) の順に選択します。

8. [Configure advanced settings] (詳細設定) ページはデフォルト設定のままにし、[Next] (次へ) を選択します。

9. [Review] (レビュー) ページで、[Deploy] (デプロイ) を選択します。

TensorFlow Lite イメージ分類コンポーネントを設定してデプロイするには (AWS CLI)

1. TensorFlow Lite イメージ分類コンポーネントのデプロイ設定を定義する deployment.json ファイルを作成します。このファイルは次のようになります:

```
{
  "targetArn":"targetArn",
  "components": {
    "aws.greengrass.TensorFlowLiteImageClassification": {
      "componentVersion": 2.1.0,
      "configurationUpdate": {
        "InferenceInterval": "60",
        "UseCamera": "true"
      }
    }
  }
}
```

   - [targetArn] フィールドで *targetArn* をデプロイメントの対象となるモノまたはモノのグループの Amazon リソースネーム (ARN) に置き換えます。形式は以下のとおりです:
     - モノ: arn:aws:iot:*region*:*account-id*:thing/*thingName*
     - モノのグループ: arn:aws:iot:*region*:*account-id*:thinggroup/*thingGroupName*
   - このチュートリアルでは、コンポーネントバージョン 2.1.0 を使用します。aws.greengrass.TensorFlowLiteImageClassification コンポーネントオブジェクトで *2.1.0* を置き換えて、別のバージョンの TensorFlow Lite イメージ分類コンポーネントを使用します。

   この設定更新により、コンポーネントはデバイスのカメラモジュールにアクセスし、カメラで撮影した画像の推論を実行します。推論コードは 60 秒ごとに実行されます。次の値の置き換え

2. 次のコマンドを実行して、デバイスに TensorFlow Lite イメージ分類コンポーネントをデプロイします。

```
aws greengrassv2 create-deployment \
    --cli-input-json file://path/to/deployment.json
```

デプロイには数分かかる場合があります。次のステップでは、コンポーネントログをチェックして、デプロイが正常に完了したことを確認し、推論結果を表示します。

ステップ 4: 推論結果の確認

コンポーネントをデプロイした後、Greengrass コアデバイスのコンポーネントログと、AWS IoT コンソールの AWS IoT MQTT クライアントで推論結果を確認できます。コンポーネントが推論結果をパブリッシュするトピックへサブスクライブするには、「ステップ 2: デフォルトの通知トピックのサブスクライブ状況を確認」を参照してください。

- AWS IoT MQTT クライアント - 推論コンポーネントがデフォルトの通知トピックにパブリッシュした結果を確認するには、次のステップを実行します:

  1. AWS IoT コンソールのナビゲーションメニューで、[Test, MQTT test client] (テスト、MQTT テストクライアント) を選択します。
  2. [Subscriptions] (サブスクリプション) 内で、**ml/tflite/image-classification** を選択します。
- コンポーネントログ - コンポーネントログで推論結果を確認するには、Greengrass コアデバイスで次のコマンドを実行します。

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.TensorFlowLiteImageClassification.log
```

コンポーネントログまたは MQTT クライアントで推論結果が確認できない場合、デプロイが失敗しているか、コアデバイスに到達していません。これは、コアデバイスがインターネットに接続されていない、あるいはコンポーネントを実行するために必要な許可がない場合に発生します。コアデバイスで次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアのログファイルを確認します。このファイルは、Greengrass コアデバイスのデプロイサービスからのログが含まれます。

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

詳細については、「機械学習の推論に対するトラブルシューティング」を参照してください。

次のステップ

このチュートリアルでは、 TensorFlow Lite イメージ分類コンポーネントと、カメラで撮影したイメージのサンプルイメージ分類を実行するカスタム設定オプションを使用する方法を示します。

パブリックコンポーネント設定のカスタマイズ、またはカスタム機械学習コンポーネントの作成の詳細については、「機械学習コンポーネントのカスタマイズ」を参照してください。

コンポーネント

AWS IoT Greengrass コンポーネントは、Greengrass のコアデバイスにデプロイするソフトウェアモジュールです。コンポーネントは、アプリケーション、ランタイムインストーラ、ライブラリ、またはデバイスで実行する任意のコードを表します。他のコンポーネントに依存するコンポーネントを定義することができます。例えば、Python をインストールするコンポーネントを定義して、その後そのコンポーネントを Python アプリケーションを実行するコンポーネントの依存関係として定義することができます。コンポーネントをデバイスのフリートにデプロイすると、Greengrass はデバイスに必要なソフトウェアモジュールのみをデプロイします。

トピック

AWS が提供したコンポーネント

AWS IoT Greengrass は、デバイスにデプロイできる構築済みコンポーネントを提供および維持します。これらのコンポーネントには、機能 (ストリームマネージャーなど)、 AWS IoT Greengrass V1 コネクタ (CloudWatch メトリクスなど)、ローカル開発ツール (AWS IoT Greengrass CLI など) が含まれます。スタンドアロンの機能性のため、デバイスにこれらのコンポーネントをデプロイ、あるいはカスタム Greengrass コンポーネントに従属関係として使用することができます。

> **Note**
>
> AWSが提供するいくつかのコンポーネントは、Greengrass nucleus の特定のマイナーバージョンによって異なります。この従属関係により、Greengrass nucleus を新しいマイナーバージョンに更新するとき、これらのコンポーネントを更新する必要があります。各コンポーネントが依存する nucleus の特定バージョンの情報については、対応するコンポーネントのトピックを参照してください。nucleus の更新の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネントのコンポーネントタイプが汎用と Lambda の両方である場合、コンポーネントの現在のバージョンは汎用タイプであり、コンポーネントの以前のバージョンは Lambda タイプです。

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
Greengrass nucleus	AWS IoT Greengrass Core ソフトウェアの nucleus。このコンポーネントを使用して、コアデバイスのソフトウェアを設定と更新します。	nucleus	Linux、Windows	はい
クライアントデバイス認証	クライアントデバイスと呼ばれるローカル IoT デバイスがコアデバイスに接続できるようにします。	プラグイン	Linux、Windows	はい
CloudWatch メトリクス	カスタムメトリクスを Amazon に発行します CloudWatch。	ジェネリック、Lambda	Linux、Windows	はい
AWS IoT Device Defender	Greengrass コアデバイスの状態の変化を管理者に通知	ジェネリック、Lambda	Linux、Windows	はい

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
	し、異常な行動を特定します。			
ディスクスプーラ	Greengrass コアデバイスから AWS IoT Core にスプールされたメッセージについて永続ストレージオプションを有効にします。このコンポーネントは、これらの送信メッセージをディスクに保存します。	プラグイン	Linux、Windows	はい
Docker アプリケーションマネージャー	AWS IoT Greengrass が Docker Hub および Amazon Elastic Container Registry (Amazon ECR) から Docker イメージをダウンロードできるようにします。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
Kinesis Video Streams 向けのエッジコネクタ	ローカルカメラからビデオフィードを読み取り、ストリームを Kinesis Video Streams に公開し、 を使用して Grafana ダッシュボードにストリームを表示します AWS IoT TwinMaker。	ジェネリック	Linux	いいえ
Greengrass CLI	ローカルなデプロイを作成して Greengrass コアデバイスとそのコンポーネントとやり取りするために使用できるコマンドラインインターフェイスを提供します。	プラグイン	Linux、Windows	はい

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
IP ディテクター	MQTT ブローカーの接続情報を に報告し AWS IoT Greengrass、クライアントデバイスが接続方法を検出できるようにします。	プラグイン	Linux、Windows	はい
Firehose	Amazon Data Firehose 配信ストリームを介して、 の送信先にデータを公開します AWS クラウド。	Lambda	Linux	いいえ
Lambda ランチャー	Lambda 関数のプロセスと環境設定を処理します。	ジェネリック	Linux	いいえ
Lambda マネージャー	Lambda 関数のプロセス間通信とスケーリングを処理します。	プラグイン	Linux	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
Lambda ランタイム	各 Lambda ランタイムにアーティファクトを提供します。	ジェネリック	Linux	いいえ
レガシーサブスクリプションルーター	AWS IoT Greengrass V1 で実行される Lambda 関数のサブスクリプションを管理します。	ジェネリック	Linux	いいえ
ローカルデバッグコンソール	Greengrass コアデバイスとそのコンポーネントのデバッグと管理に使用できるローカルコンソールを提供します。	プラグイン	Linux、Windows	はい
ログマネージャー	Greengrass コアデバイス上にログを収集してアップロードします。	プラグイン	Linux、Windows	はい

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
機械学習コンポーネント	Greengrass コアデバイスで機械学習の推論を実行するために使用できる機械学習モデルとサンプル推論コードを提供します。	機械学習コンポーネント を参照してください。		
Modbus-RTU プロトコルアダプタ	ローカルの Modbus RTU デバイスから情報をポーリングします。	Lambda	Linux	いいえ
nucleus テレメトリエミッタ	nucleus から収集されたシステムヘルステレメトリデータをローカルトピックまたは AWS IoT Core MQTT トピックに発行します。	プラグイン	Linux、Windows	はい

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
MQTT ブリッジ	クライアントデバイス、ローカルパ AWS IoT Greengrass ブリッシュ/サブスクライブ、および間で MQTT メッセージをリレーします AWS IoT Core。	プラグイン	Linux、Windows	はい
MQTT 3.1.1 ブローカー (モケット)	クライアントデバイスとコアデバイスの間のメッセージを処理する、MQTT 3.1.1 ブローカーを実行します。	プラグイン	Linux、Windows	はい
MQTT 5 ブローカー (EMQX)	クライアントデバイスとコアデバイスの間のメッセージを処理する、MQTT 5 ブローカーを実行します。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
PKCS#11 プロバイダ	Greengrass コンポーネントに対して、ハードウェアセキュリティモジュール (HSM) に安全に保存しているプライベートキーと証明書へのアクセスを有効にします。	プラグイン	Linux	はい
シークレットマネージャー	Greengrass コアデバイスのカスタムコンポーネントでパスワードなどの認証情報を安全に使用できるように、シークレットから AWS Secrets Manager シークレットをデプロイします。	プラグイン	Linux、Windows	はい

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
セキュアトンネリング	制限されたファイアウォールの背後にある Greengrass コアデバイスとの双方向通信を確立するために使用できる AWS IoT セキュアトンネリング接続を有効にします。	ジェネリック	Linux	いいえ
シャドウマネージャー	コアデバイス上のシャドウとの対話を有効にします。シャドウドキュメントストレージと、ローカルシャドウ状態の AWS IoT Device Shadow サービスとの同期も管理します。	プラグイン	Linux、Windows	はい

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
Amazon SNS	Amazon SNS トピックにメッセージを公開します。	Lambda	Linux	いいえ
ストリームマネージャー	大量のデータをローカルソースから AWS クラウドにストリーミングします。	ジェネリック	Linux、Windows	いいえ
Systems Manager エージェント	を使用してコアデバイスを管理します。これにより AWS Systems Manager、デバイスにパッチを適用したり、コマンドを実行したりできます。	ジェネリック	Linux	いいえ
トークン交換サービス	サービスとやり取り AWS するために使用できる AWS 認証情報を提供します。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
IoT SiteWise OPC-UA コレクター	OPC-UA サーバーからデータを収集します。	ジェネリック	Linux、Windows	いいえ
IoT SiteWise OPC-UA データソースシミュレーター	サンプルデータを生成するローカル OPC-UA サーバーを実行します。	ジェネリック	Linux、Windows	いいえ
IoT SiteWise パブリッシャー	AWS クラウドにデータを発行します。	ジェネリック	Linux、Windows	いいえ
IoT SiteWise プロセッサ	Greengrass コアデバイス上にデータを処理します。	ジェネリック	Linux、Windows	いいえ

Greengrass nucleus

Greengrass nucleus コンポーネント (`aws.greengrass.Nucleus`) は必須コンポーネントであり、デバイスで AWS IoT Greengrass Core ソフトウェアを実行するための最小要件です。このコンポーネントを設定して、 AWS IoT Greengrass Core ソフトウェアをリモートでカスタマイズおよび更新できます。このコンポーネントをデプロイして、プロキシ、デバイスロール、 AWS IoT モノの設定などの設定をコアデバイスに構成します。

> **Important**
>
> nucleus コンポーネントのバージョンが変更されたり、特定の設定パラメータを変更したりすると、nucleus およびデバイス上の他のすべてのコンポーネントを含む AWS IoT Greengrass Core ソフトウェアが再起動して変更を適用します。

コンポーネントをデプロイすると、 はそのコンポーネントのすべての依存関係の最新のサポート対象バージョン AWS IoT Greengrass をインストールします。このため、モノのグループに新しいデバイスを追加したり、それらのデバイスを対象とするデプロイを更新したりすると、 AWSが提供するパブリックコンポーネントの新しいパッチバージョンがコアデバイスに自動的にデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。
デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。
AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「」を参照してくださいAWS IoT Greengrass Core ソフトウェア (OTA) の更新。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.12.x
- 2.11.x
- 2.10.x
- 2.9.x
- 2.8.x
- 2.7.x
- 2.6.x

- 2.5.x
- 2.4.x
- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

詳細については、「サポートされているプラットフォーム」を参照してください。

要件

Greengrass nucleus と AWS IoT Greengrass Core ソフトウェアをインストールして実行するには、デバイスが特定の要件を満たしている必要があります。詳細については、「デバイスの要件」を参照してください。

Greengrass nucleus コンポーネントは VPC での実行がサポートされています。このコンポーネントを VPC にデプロイするには、以下が必要です。

- Greengrass nucleus コンポーネントには、 AWS IoT data、 AWS IoT 認証情報、および Amazon S3 への接続が必要です。

依存関係

Greengrass nucleus にはコンポーネントの依存関係は含まれません。ただし、いくつかの AWS で提供されるコンポーネントには、依存関係としての nucleus が含まれます。詳細については、「AWS が提供したコンポーネント」を参照してください。

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

ダウンロードとインストール

お使いのデバイスに Greengrass nucleus コンポーネントを設定するインストーラをダウンロードできます。このインストーラは、お使いのデバイスを Greengrass コアデバイスとしてセットアップします。実行できるインストールには、必要な AWS リソースを作成するクイックインストールと、自分で AWS リソースを作成する手動インストールの 2 種類があります。詳細については、「AWS IoT Greengrass Core ソフトウェアをインストールします。」を参照してください。

チュートリアルに従って Greengrass nucleus をインストールし、Greengrass コンポーネントの開発を参照することもできます。詳細については、「チュートリアル: AWS IoT Greengrass V2 の開始方法」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。一部のパラメータでは、 AWS IoT Greengrass Core ソフトウェアを再起動して有効にする必要があります。このコンポーネントの設定理由と方法の詳細については、「AWS IoT Greengrass Core ソフトウェアを設定する」を参照してください。

`iotRoleAlias`

トークン交換 IAM AWS IoT ロールを指すロールエイリアス。 AWS IoT 認証情報プロバイダーはこのロールを引き受けて、Greengrass コアデバイスが AWS サービスとやり取りできるようにします。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

`--provision true` オプションを指定して AWS IoT Greengrass Core ソフトウェアを実行すると、ソフトウェアはロールエイリアスをプロビジョニングし、nucleus コンポーネントにその値を設定します。

`interpolateComponentConfiguration`

(オプション) コンポーネント設定のコンポーネント recipe 変数、およびマージ設定の更新を補間するように、Greengrass nucleus を有効化することができます。コアデバイスが、recipe 変数を設定内で使用する Greengrass コンポーネントを実行できるように、このオプションを `true` に設定することをお勧めします。

この機能は、このコンポーネントの v2.6.0 以降で利用可能です。

デフォルト: `false`

`networkProxy`

(オプション) すべての接続に使用するネットワークプロキシ。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

> **Important**
>
> この設定パラメータに変更をデプロイすると、 AWS IoT Greengrass Core ソフトウェアが再起動して変更が有効になります。

このオブジェクトには、次の情報が含まれます。

`noProxyAddresses`

(オプション) プロキシの対象外となる IP アドレスやホスト名のカンマ区切りリスト。

`proxy`

接続先のプロキシ。このオブジェクトには、次の情報が含まれます。

`url`

プロキシサーバーの URL (`scheme://userinfo@host:port` 形式)。

- `scheme` - スキーム。`http` または `https` である必要があります。

  > **Important**
  >
  > HTTPS プロキシを使用するには、Greengrass コアデバイスで Greengrass nucleus v2.5.0 以降を実行している必要があります。
  > HTTPS プロキシを設定する場合は、コアデバイスの Amazon ルート CA 証明書にプロキシサーバー CA 証明書を追加する必要があります。詳細については、「コアデバイスが HTTPS プロキシを信頼できるようにする」を参照してください。

- `userinfo` - (オプション) ユーザー名とパスワードの情報。この情報を `url` で指定する場合、Greengrass コアデバイスは `username` および `password` フィールドを無視します。
- `host` – プロキシサーバーのホスト名または IP アドレス。
- `port` - (オプション) ポート番号。ポートを指定しない場合、Greengrass コアデバイスは次のデフォルト値を使用します。

- `http` – 80
- `https` – 443

`username`

(オプション) プロキシサーバーを認証するユーザー名です。

`password`

(オプション) プロキシサーバーを認証するパスワードです。

`mqtt`

(オプション) Greengrass コアデバイスの MQTT 設定。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

> **Important**
>
> この設定パラメータに変更をデプロイすると、 AWS IoT Greengrass Core ソフトウェアが再起動して変更が有効になります。

このオブジェクトには、次の情報が含まれます。

`port`

(オプション) MQTT 接続に使用するポート。

デフォルト: `8883`

`keepAliveTimeoutMs`

(オプション) MQTT 接続を維持するためにクライアントが送信する各 `PING` メッセージを送信する間隔 (ミリ秒)。この値は `pingTimeoutMs` より大きくなければなりません。

デフォルト: `60000` (60 秒)

`pingTimeoutMs`

(オプション) クライアントがサーバーから `PINGACK` メッセージを受信するまで待機する時間 (ミリ秒)。待機時間がタイムアウトを超えると、コアデバイスは MQTT 接続を閉じて再度開きます。この値は、`keepAliveTimeoutMs` より小さくなければなりません。

デフォルト: `30000` (30 秒)

`operationTimeoutMs`

(オプション) MQTT 操作 (`CONNECT` や `PUBLISH` など) が完了するまでクライアントが待機する時間 (ミリ秒)。このオプションは MQTT `PING` メッセージやキープアライブメッセージには適用されません。

デフォルト: `30000` (30 秒)

`maxInFlightPublishes`

(オプション) 同時に送信できる未確認の MQTT QoS 1 メッセージの最大数。

この機能は、このコンポーネントの v2.1.0 以降に利用できます。

デフォルト: `5`

有効な範囲: 最大値は 100 です。

`maxMessageSizeInBytes`

(オプション) MQTT メッセージの最大サイズ。メッセージがこのサイズを超えると、Greengrass nucleus でエラーとなりメッセージが拒否されます。

この機能は、このコンポーネントの v2.1.0 以降に利用できます。

デフォルト: `131072` (128 KB)

有効な範囲: 最大値は `2621440` (2.5 MB) です。

`maxPublishRetry`

(オプション) パブリッシュに失敗したメッセージを再試行する最大回数。`-1` の再試行回数の上限はありません。

この機能は、このコンポーネントの v2.1.0 以降に利用できます。

デフォルト: `100`

`spooler`

(オプション) Greengrass コアデバイスの MQTT スプーラ設定。このオブジェクトには、次の情報が含まれます。

`storageType`

メッセージを保存するためのストレージタイプ。`storageType` が `Disk` に設定されている場合、`pluginName` を設定できます。`Memory` または `Disk` のどちらかを指定できます。

この機能は、Greengrass nucleus コンポーネントの v2.11.0 以降で利用できます。

> **Important**
>
> MQTT スプーラの `storageType` が `Disk` に設定されていて、Greengrass nucleus をバージョン 2.11.x から以前のバージョンにダウングレードしたい場合は、設定を `Memory` に戻す必要があります。2.10.x 以前のバージョンの Greengrass nucleus でサポートされている `storageType` の設定は `Memory` のみです。このガイダンスに従わないと、スプーラが壊れる可能性があります。スプーラが壊れると、Greengrass コアデバイスが MQTT メッセージを AWS クラウド に送信できなくなります。

デフォルト: `Memory`

`pluginName`

(オプション) プラグインコンポーネント名。このコンポーネントは、`storageType` が `Disk` に設定されている場合にのみ使用されます。このオプションのデフォルトは `aws.greengrass.DiskSpooler` で、Greengrass が提供するディスクスプーラを使用します。

この機能は、Greengrass nucleus コンポーネントの v2.11.0 以降で利用できます。

デフォルト: `"aws.greengrass.DiskSpooler"`

`maxSizeInBytes`

(オプション) コアデバイスが未処理の MQTT メッセージをメモリに格納するキャッシュの最大サイズ。キャッシュがいっぱいになると、新しいメッセージは拒否されます。

デフォルト: `2621440` (2.5 MB)

`keepQos0WhenOffline`

(オプション) コアデバイスがオフライン中に受信する MQTT QoS 0 メッセージをスプールできます。このオプションを `true` に設定した場合、コアデバイスは、オフライン中に送

信できない QoS 0 メッセージをスプールします。このオプションを `false` に設定した場合、コアデバイスはこれらのメッセージを廃棄します。コアデバイスは、スプールがいっぱいでない限り、常に QoS 1 メッセージをスプールします。

デフォルト: `false`

`version`

(オプション) MQTT のバージョン。`mqtt3` または `mqtt5` のどちらかを指定できます。

この機能は、Greengrass nucleus コンポーネントの v2.10.0 以降で利用できます。

デフォルト: `mqtt5`

`receiveMaximum`

(オプション) ブローカーが送信できる未承認の QoS1 パケットの最大数。

この機能は、Greengrass nucleus コンポーネントの v2.10.0 以降で利用できます。

デフォルト: `100`

`sessionExpirySeconds`

(オプション) IoT Core からのセッション継続を要求できる時間 (秒) を指定します。デフォルトは、 でサポートされる最大時間です AWS IoT Core。

この機能は、Greengrass nucleus コンポーネントの v2.10.0 以降で利用できます。

デフォルト: `604800 (7 days)`

`minimumReconnectDelaySeconds`

(オプション) 再接続動作のオプション。MQTT が再接続するまでの最小時間 (秒) を指定します。

この機能は、Greengrass nucleus コンポーネントの v2.10.0 以降で利用できます。

デフォルト: `1`

`maximumReconnectDelaySeconds`

(オプション) 再接続動作のオプション。MQTT が再接続するまでの最大時間 (秒) を指定します。

この機能は、Greengrass nucleus コンポーネントの v2.10.0 以降で利用できます。

デフォルト: `120`

`minimumConnectedTimeBeforeRetryResetSeconds`

(オプション) 再接続動作のオプション。リトライ遅延が最小値にリセットされる以前に、接続がアクティブでなければならない時間 (秒) を指定します。

この機能は、Greengrass nucleus コンポーネントの v2.10.0 以降で利用できます。

デフォルト: `30`

`jvmOptions`

(オプション) AWS IoT Greengrass Core ソフトウェアの実行に使用する JVM オプション。AWS IoT Greengrass Core ソフトウェアの実行に推奨される JVM オプションについては、「」を参照してくださいJVM オプションでメモリ割り当てを制御する。

> **Important**
>
> この設定パラメータに変更をデプロイすると、 AWS IoT Greengrass Core ソフトウェアが再起動して変更が有効になります。

`iotDataEndpoint`

AWS IoT のデータエンドポイント AWS アカウント。

`--provision true` オプションを指定して AWS IoT Greengrass Core ソフトウェアを実行すると、ソフトウェアは からデータと認証情報エンドポイントを取得し AWS IoT 、nucleus コンポーネントに設定します。

`iotCredEndpoint`

の AWS IoT 認証情報エンドポイント AWS アカウント。

`--provision true` オプションを指定して AWS IoT Greengrass Core ソフトウェアを実行すると、ソフトウェアは からデータと認証情報エンドポイントを取得し AWS IoT 、nucleus コンポーネントに設定します。

`greengrassDataPlaneEndpoint`

この機能は、このコンポーネントの v2.7.0 以降で利用可能です。

詳細については、「プライベート CA によって署名されたデバイス証明書を使用する」を参照してください。

`greengrassDataPlanePort`

この機能は、このコンポーネントの v2.0.4 以降で利用できます。

(オプション) データプレーン接続に使用するポート。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

> **Important**
>
> デバイスがアウトバウンド接続を確立できるポートを指定する必要があります。ブロックされているポートを指定すると、デバイスは に接続 AWS IoT Greengrass してデプロイを受信できなくなります。

次のオプションから選択します。

- `443`
- `8443`

デフォルト: `8443`

`awsRegion`

AWS リージョン 使用する 。

`runWithDefault`

コンポーネントの実行に使用するシステムユーザー。

> **Important**
>
> この設定パラメータに変更をデプロイすると、 AWS IoT Greengrass Core ソフトウェアが再起動して変更が有効になります。

このオブジェクトには、次の情報が含まれます。

`posixUser`

コアデバイスがジェネリックコンポーネントおよび Lambda コンポーネントを実行するために使用するシステムユーザーの名前または ID (オプション)。ユーザーとグループを `user:group` の形式に従ってコロン (`:`) で区切って指定します。グループはオプションで

す。グループを指定しない場合、 AWS IoT Greengrass Core ソフトウェアはユーザーのプライマリグループを使用します。たとえば、`ggc_user` や `ggc_user:ggc_group` と指定することができます。詳細については、「コンポーネントを実行するユーザーを設定する」を参照してください。

`--component-default-user` *`ggc_user`*`:`*`ggc_group`* オプションを指定して AWS IoT Greengrass Core ソフトウェアインストーラを実行すると、ソフトウェアはこのパラメータを nucleus コンポーネントに設定します。

`windowsUser`

この機能は、このコンポーネントの v2.5.0 以降で利用できます。

Windows コアデバイスでこのコンポーネントを実行するために使用する Windows ユーザーの名前。ユーザーは各 Windows コアデバイスに存在し、その名前とパスワードは LocalSystem アカウントの認証情報マネージャーインスタンスに保存されている必要があります。詳細については、「コンポーネントを実行するユーザーを設定する」を参照してください。

`--component-default-user` *`ggc_user`* オプションを指定して AWS IoT Greengrass Core ソフトウェアインストーラを実行すると、ソフトウェアはこのパラメータを nucleus コンポーネントに設定します。

`systemResourceLimits`

この機能は、このコンポーネントの v2.4.0 以降に利用できます。 AWS IoT Greengrass は、現在 Windows コアデバイスにこの機能をサポートしていません。

デフォルトで、ジェネリックおよびコンテナ化されていない Lambda コンポーネントプロセスに適用されるシステムリソースの制限。デプロイを作成するときに、個々のコンポーネントのシステムリソース制限を上書きできます。詳細については、「コンポーネントのシステムリソース制限を設定する」を参照してください。

このオブジェクトには、次の情報が含まれます。

`cpus`

各コンポーネントのプロセスがコアデバイスで使用できる CPU 時間の最大量。コアデバイスの合計 CPU 時間は、デバイスの CPU コア数と同じです。たとえば、4 つの CPU コアを持つコアデバイスでは、この値を 2 に設定することで、各コンポーネントのプロセスを各 CPU コアの 50% の使用率に制限することができます。CPU コアが 1 つのデバイスの場合は、この値を `0.25` に設定することで、各コンポーネントのプロセスを CPU の 25% の使用率に制限することができます。この値を CPU コア数よりも大きい数値に設定

した場合、 AWS IoT Greengrass Core ソフトウェアはコンポーネントの CPU 使用率を制限しません。

`memory`

各コンポーネントのプロセスがコアデバイスで使用できる RAM の最大量 (キロバイト単位)。

`s3EndpointType`

(オプション) S3 エンドポイントタイプ。このパラメータは、米国東部 (バージニア北部) (`us-east-1`) リージョンでのみ有効です。他のリージョンからこのパラメータを設定しても無視されます。次のオプションから選択します。

- `REGIONAL` – S3 クライアントと署名付き URL はリージョンエンドポイントを使用します。
- `GLOBAL` – S3 クライアントと署名付き URL はレガシーエンドポイントを使用します。

デフォルト: `GLOBAL`

`logging`

(オプション) コアデバイスのログ設定。Greengrass ログの設定と使用方法の詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

このオブジェクトには、次の情報が含まれます。

`level`

(オプション) 出力するログメッセージの最小レベル。

こちらにレベル順に一覧表示されているログレベルから選択します。

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

デフォルト: `INFO`

`format`

(オプション) ログのデータ形式。次のオプションから選択します。

- `TEXT` - ログをテキスト形式で表示する場合は、このオプションを選択します。
- `JSON` - Greengrass CLI のログコマンドでログを表示したり、プログラムでログとやり取りしたりするには、このオプションを選択します。

デフォルト: TEXT

`outputType`

(オプション) ログの出力タイプ。次のオプションから選択します。

- FILE – AWS IoT Greengrass Core ソフトウェアは、 で指定したディレクトリ内のファイルにログを出力しますoutputDirectory。
- CONSOLE — AWS IoT Greengrass Core ソフトウェアはログを に出力しますstdout。コアデバイスがログを印刷するときにログを表示するには、このオプションを選択します。

デフォルト: FILE

`fileSizeKB`

(オプション) 各ログファイルの最大サイズ (キロバイト単位)。ログファイルがこの最大ファイルサイズを超えると、 AWS IoT Greengrass Core ソフトウェアは新しいログファイルを作成します。

このパラメータは、FILE または outputType が指定されている場合にのみ適用されます。

デフォルト: 1024

`totalLogsSizeKB`

(オプション) Greengrass nucleus を含む各コンポーネントのログファイルの最大合計サイズ (キロバイト単位)。Greengrass nucleus のログファイルには、プラグインコンポーネントからのログも含まれます。コンポーネントのログファイルの合計サイズがこの最大サイズを超えると、 AWS IoT Greengrass Core ソフトウェアはそのコンポーネントの最も古いログファイルを削除します。

このパラメータは、ログマネージャーコンポーネントのディスク容量の制限パラメータ (diskSpaceLimit) と同等で、Greengrass nucleus (システム) と各コンポーネントに対して指定できます。 AWS IoT Greengrass Core ソフトウェアは、Greengrass nucleus と各コンポーネントの最大合計ログサイズとして 2 つの値の最小値を使用します。

このパラメータは、FILE または outputType が指定されている場合にのみ適用されます。

デフォルト: 10240

`outputDirectory`

(オプション) ログファイルの出力ディレクトリ。

このパラメータは、FILE または outputType が指定されている場合にのみ適用されます。

デフォルト: */greengrass/v2*/logs。 */greengrass/v2*は AWS IoT Greengrass ルートフォルダです。

`fleetstatus`

このパラメータは、このコンポーネントの v2.1.0 以降で利用できます。

(オプション) コアデバイスのフリートステータス設定。

このオブジェクトには、次の情報が含まれます。

`periodicStatusPublishIntervalSeconds`

(オプション) コアデバイスがデバイスステータスを AWS クラウド にパブリッシュする時間 (秒単位)。

最小:86400 (24 時間)

デフォルト: 86400 (24 時間)

`telemetry`

(オプション) コアデバイスのシステムヘルステレメトリ設定。テレメトリメトリクスとテレメトリデータに対する動作の詳細については、「AWS IoT Greengrass コアデバイスからシステムヘルステレメトリデータを収集する」を参照してください。

このオブジェクトには、次の情報が含まれます。

`enabled`

(オプション) テレメトリを有効または無効にできます。

デフォルト: `true`

`periodicAggregateMetricsIntervalSeconds`

(オプション) コアデバイスがメトリクスを集約する間隔 (秒単位)。

この値をサポートされている最小値よりも低く設定すると、nucleus では代わりにデフォルト値が使用されます。

最小: 3600

デフォルト: 3600

`periodicPublishMetricsIntervalSeconds`

(オプション) コアデバイスがテレメトリメトリクスを AWS クラウド にパブリッシュする間隔 (秒単位)。

この値をサポートされている最小値よりも低く設定すると、nucleus では代わりにデフォルト値が使用されます。

最小: `86400`

デフォルト: `86400`

`deploymentPollingFrequencySeconds`

(オプション) デプロイ通知をポーリングする期間 (秒)。

デフォルト: `15`

`componentStoreMaxSizeBytes`

(オプション) コンポーネントの recipe とアーティファクトを含む、コンポーネントストアのディスク上の最大サイズ。

デフォルト: `10000000000` (10 GB)

`platformOverride`

(オプション) コアデバイスのプラットフォームを識別する属性のディクショナリ。これを使用して、コンポーネント recipe がコンポーネントの正しいライフサイクルとアーティファクトを識別するために使用できるカスタムプラットフォーム属性を定義します。たとえば、ハードウェア機能属性を定義して、コンポーネントを実行するアーティファクトの最小セットのみをデプロイできます。詳細については、コンポーネント recipe の「マニフェストプラットフォームパラメータ」を参照してください。

また、このパラメータを使用して、コアデバイスの `os` および `architecture` プラットフォーム属性を上書きすることができます。

`httpClient`

このパラメータは、このコンポーネントの v2.5.0 以降で利用できます。

(オプション) コアデバイスの HTTP クライアント設定。これらの設定オプションは、このコンポーネントによって行われたすべての HTTP リクエストに適用されます。コアデバイスが低速の

ネットワーク上で動作している場合、これらのタイムアウト時間を長くして HTTP 要求がタイムアウトするのを防ぐことができます。

このオブジェクトには、次の情報が含まれます。

connectionTimeoutMs

(オプション) 接続を開いた際に接続要求がタイムアウトするまの待機時間 (ミリ秒)。

デフォルト: 2000 (2 秒)

socketTimeoutMs

(オプション) 開いている接続でデータを転送した際に接続がタイムアウトするまでの待機時間 (ミリ秒)。

デフォルト: 30000 (30 秒)

Example 例: 設定マージの更新

```
{
  "iotRoleAlias": "GreengrassCoreTokenExchangeRoleAlias",
  "networkProxy": {
    "noProxyAddresses": "http://192.168.0.1,www.example.com",
    "proxy": {
      "url": "http://my-proxy-server:1100",
      "username": "Mary_Major",
      "password": "pass@word1357"
    }
  },
  "mqtt": {
    "port": 443
  },
  "greengrassDataPlanePort": 443,
  "jvmOptions": "-Xmx64m",
  "runWithDefault": {
    "posixUser": "ggc_user:ggc_group"
  }
}
```

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* を AWS IoT Greengrass ルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.12.2	バグ修正と機能向上 • 古いログが正しくクリーンアップされない問題を修正しました。 • 一般的なバグ修正と機能強化。
2.12.1	バグ修正と機能向上 • nucleus がデプロイトピックへの MQTT サブスクリプションを複製して、追加のログ記録と MQTT 発行が発生する可能性がある問題を修正しました。

バージョン	変更
2.12.0	新機能 • ロールバックデプロイの一部としてブートストラップライフサイクルステップを実行できます。
2.11.3	バグ修正と機能向上 • nucleus の依存関係が失敗したときにコンポーネントが不適切に開始される可能性がある問題を修正しました。 新機能 • 設定可能な S3 エンドポイントタイプを追加します。
2.11.2	バグ修正と機能向上 • 多数 (50 超) のサブスクリプションが使用されている場合にオフラインとして表示される可能性がある nucleus MQTT 5 クライアントの問題を修正します。 • Docker ダイヤル TCP エラーについての再試行を追加します。
2.11.1	バグ修正と機能向上 • ブートストラップタスクが失敗し、デプロイメタデータファイルが破損している場合に nucleus が起動しない問題を修正します。 • オンデマンド Lambda コンポーネントがデプロイステータスの更新で報告されない問題を修正します。 • 重複する認可ポリシー ID のサポートを追加します。
2.11.0	新機能 • ローカルのデプロイをキャンセルできます。 • ローカルのデプロイの障害処理ポリシーを設定できます。 • ディスクスプーラプラグインのサポートを追加します。
2.10.3	バグ修正と機能向上 • PKCS #11 プロバイダーを使用しているときに Greengrass がデプロイ通知をサブスクライブしない問題を修正しました。

バージョン	変更
2.10.2	バグ修正と機能向上 • コンポーネントのライフサイクルの大文字と小文字を区別しない解析が可能になります。 • 環境 PATH 変数が正しく再作成されない問題を修正しました。 • 特殊文字を含むユーザー名のストリームマネージャーを含むコンポーネントのプロキシ URI エンコーディングを修正しました。
2.10.1	バグ修正と機能向上 • Jetson Nano など、特定の ARMv8 プロセッサで起動時にクラッシュする可能性がある問題を修正しました。 • Greengrass はコンポーネントの標準入力を閉じないようにしました。これにより、2.10.0 以前の動作に戻ります。

バージョン	変更
2.10.0	新機能 • 空の正規表現に `interpolateComponentConfiguration` サポートを追加します。Greengrass はルート設定オブジェクトから補間するようになりました。 • MQTT5 のサポートを追加します。 • プラグインコンポーネントをスキャンせずに素早く読み込むメカニズムを追加します。 • Greengrass が未使用の Docker イメージを削除することでディスクスペースを節約できるようにします。 バグ修正と機能向上 • ロールバック時に、特定の設定値がデプロイからそのままになる問題を修正します。 • Greengrass nucleus がカスタムのAWS 非認証情報およびデータエンドポイントで AWS ドメインシーケンスを検証する問題を修正しました。 • マルチグループの依存関係解決を更新して、アクティブなバージョンにロックするのではなく、 AWS クラウド ネゴシエーションによってすべてのグループの依存関係を再解決します。この更新により、デプロイエラーコード `INSTALLED_COMPONENT_NOT_FOUND` も削除されます。 • Greengrass nucleus を更新し、既にローカルに存在する Docker イメージのダウンロードをスキップするようにしました。 • Greengrass nucleus を更新し、タイムアウトが切れる前にコンポーネントのインストールステップを再開するようにしました。 • 追加のマイナー修正と機能向上。
2.9.6	バグ修正と機能向上 • Greengrass のデプロイがエラー LAUNCH_DIRECTORY_CORRUPTED で失敗し、その後のデバイス再起動で Greengrass の起動に失敗する問題を修正しました。このエラーは、Greengrass の再起動が必要なデプロイで、複数のモノグループ間で Greengrass デバイスを移動した場合に発生する場合があります。

バージョン	変更
2.9.5	新機能 • Greengrass nucleus ソフトウェアの署名検証をサポートするようになりました。 バグ修正と機能向上 • ローカル recipe のメタデータ領域が Greengrass nucleus の起動領域と一致しない場合に、デプロイが失敗する問題を修正しました。Greengrass nucleus は、このような場合にクラウドと再交渉するようになりました。 • MQTT メッセージスプーラが一杯になり、メッセージが削除されない問題を修正しました。 • 追加のマイナー修正と機能向上。
2.9.4	バグ修正と機能向上 • QOS 0 メッセージを破棄する前に null メッセージの有無がチェックされます。 • ジョブステータスの詳細値は、1024 文字の制限を超えると、切り捨てられます。 • Windows のブートストラップスクリプトを更新して、パスにスペースが含まれている場合に Greengrass ルートパスを正しく読み取れるようにします。 • サブスクリプションレスポンスが送信されなかった場合にクライアントメッセージを削除する AWS IoT Core ように、 をサブスクライブするように更新します。 • メイン設定ファイルが破損しているかまたは見つからない場合に nucleus によってバックアップファイルから設定が確実に読み込まれます。

バージョン	変更
2.9.3	バグ修正と機能向上 • MQTT クライアント ID が重複しないようにします。 • ファイルの読み取りと書き込みがより堅牢になり、破損の回避と回復を図っています。 • ネットワークに関連する特定のエラーの発生時に、Docker イメージのプルを再試行するようになりました。 • MQTT 接続の `noProxyAddresses` オプションを追加します。
2.9.2	バグ修正と機能向上 • 進行中のデプロイに `interpolateComponentConfiguration` の設定が適用されない問題を修正しました。 • OSHI を使用して、すべての子プロセスを一覧表示します。
2.9.1	バグ修正と機能向上 • デプロイがプラグインコンポーネントを削除する場合に Greengrass が再起動する問題の修正を追加しました。
2.9.0	新機能 • サブデプロイを作成して、デバイスのより小さなサブセットでデプロイを再試行する機能を追加しました。この機能により、失敗したデプロイをより効率的にテストして解決できます。 バグ修正と機能向上 • `useradd`、`groupadd`、および `usermod` を持たないシステムのサポートを改善しました。 • 追加のマイナー修正と機能向上。

バージョン	変更
2.8.1	バグ修正と機能向上 • Greengrass API エラーからデプロイエラーコードが正しく生成されなかった問題を修正します。 • デプロイ中にコンポーネントが ERRORED 状態に達したときに、フリートステータスの更新によって不正確な情報が送信される問題を修正します。 • Greengrass の既存のサブスクリプションが 50 を超える場合にデプロイを完了できなかった問題を修正します。

バージョン	変更
2.8.0	新機能 • コアデバイスへのコンポーネントのデプロイで問題が発生した場合に、デプロイのヘルスステータスのレスポンス (詳細なエラーコードを含む) を報告するように、Greengrass nucleus を更新します。詳細については、「詳細なデプロイエラーコード」を参照してください。 • コンポーネントが BROKEN または ERRORED の状態になったときに、コンポーネントのヘルスステータスレスポンス (詳細なエラーコードを含む) を報告するように、Greengrass nucleus を更新します。詳細については、「詳細なコンポーネントのステータスコード」を参照してください。 • ステータスメッセージフィールドを展開して、デバイスのクラウド可用性情報を改善します。 • フリートステータスサービスの堅牢性を向上させます。 バグ修正と機能向上 • 設定が変更されたときに、壊れたコンポーネントを再インストールできるようにします。 • ブートストラップデプロイ中に nucleus が再起動するとデプロイが失敗する問題を修正します。 • ルートパスにスペースが含まれているとインストールが失敗する Windows の問題を修正します。 • デプロイ中にシャットダウンされたコンポーネントが新しいバージョンのシャットダウンスクリプトを使用する問題を修正します。 • シャットダウンのさまざまな改善。 • 追加のマイナー修正と機能向上。

バージョン	変更
2.7.0	新機能 • Greengrass nucleus を更新して、コアデバイスがローカルデプロイを適用したときにステータスの更新を AWS IoT Greengrass クラウドに送信します。 • CA が に登録されていないカスタム認証機関 (CA) によって署名されたクライアント証明書のサポートを追加します AWS IoT。この機能を使用するには、新しい `greengrassDataPlaneEndpoint` 設定オプションに `iotdata` を設定します。詳細については、「プライベート CA によって署名されたデバイス証明書を使用する」を参照してください。 バグ修正と機能向上 • Nucleus が停止または再起動された場合の特定のシナリオで、Greengrass nucleus がデプロイをロールバックする問題を修正しました。今後 Nucleus は、再起動した後にも同じデプロイで動作します。 • ソフトウェアをシステムサービスとしてセットアップする際、`--start` 引数が反映されるように Greengrass インストーラを更新しました。 • nucleus がコンポーネントを更新したイベントのデプロイ ID を設定するように、SubscribeToComponentUpdates の動作を更新しました。 • 追加のマイナー修正と機能向上。

バージョン	変更
2.6.0	新機能 • ローカルで公開/サブスクライブされるトピックをサブスクライブする際に、MQTT ワイルドカードが使用できるようになりました。詳細については、ローカルメッセージをパブリッシュ/サブスクライブするおよびSubscribeToTopicを参照してください。 • コンポーネント構成で、*component_dependency_name* :configuration: *json_pointer* recipe 変数以外の recipe 変数がサポートされました。これらの recipe 変数は、recipe でコンポーネントの DefaultConfiguration を定義している場合、またはデプロイでコンポーネントを構成している場合に使用できます。この機能を有効にするには、interpolateComponentConfiguration設定オプションを に設定しますtrue。詳細については、レシピ変数およびマージ更新で recipe 変数を使用するを参照してください。 • プロセス間通信 (IPC) 承認ポリシーでの、* ワイルドカードの完全なサポートを追加しました。これにより、リソース文字列内の * 文字を、文字の任意の組み合わせと一致させる指定が可能になりました。詳細については、「承認ポリシー内のワイルドカード」を参照してください。 • Greengrass CLI が使用する IPC オペレーションを呼び出すために、カスタムコンポーネントが使用できるようになりました。これらの IPC オペレーションにより、ローカルでのデプロイ管理、コンポーネントの詳細の表示、およびローカルのデバッグコンソールへのサインイン用のパスワード生成を行うことができます。詳細については、「IPC: Manage local deployments and components」(IPC: ローカルのデプロイとコンポーネントを管理する) を参照してください。 バグ修正と機能向上 • 特定のシナリオで、ハード依存関係が再起動もしくはステータスを変更をした際に、依存関係のあるコンポーネントが反応しない問題を修正しました。 • デプロイが失敗した場合にコアデバイスが AWS IoT Greengrass クラウドサービスに報告するエラーメッセージを改善します。 • 特定のシナリオでの Greengrass nucleus の再起動時、nucleus がモノのデプロイを 2 回適用していた問題を修正しました。

バージョン	変更
	• 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。
2.5.6	新機能 • ECC キーを使用するハードウェアセキュリティモジュールのサポートが追加されました。ハードウェアセキュリティモジュール (HSM) を使用して、デバイスのプライベートキーと証明書を安全に保存できます。詳細については、「ハードウェアセキュリティ統合」を参照してください。 バグ修正と機能向上 • 特定のシナリオで壊れたインストールスクリプトを使用してコンポーネントをデプロイすると、デプロイが完了しない問題を修正しました。 • 起動時のパフォーマンスが向上しました。 • 追加のマイナー修正と機能向上。
2.5.5	新機能 • コンポーネントの `GG_ROOT_CA_PATH` 環境変数が追加され、カスタムコンポーネントのルート認証局 (CA) 証明書にアクセスできるようになりました。 バグ修正と機能向上 • 英語以外の表示言語を使用する Windows デバイスのサポートが追加されました。 • Greengrass nucleus がインストーラ引数のブール値を解析する方法が更新され、`true` 値を指定するブール値なしでブール引数を指定できるようになりました。たとえば、`--provision true` の代わりに `--provision` をクリックして、自動リソースプロビジョニングを使用してインストールできるようになりました。 • 特定のシナリオでプロビジョニングした後に、コアデバイスがステータスを AWS IoT Greengrass クラウドサービスに報告しない問題を修正しました。 • 追加のマイナー修正と機能向上。

バージョン	変更
2.5.4	バグ修正と機能向上 • 一般的なバグ修正と機能強化。
2.5.3	新機能 • ハードウェアセキュリティ統合のサポートが追加されました。ハードウェアセキュリティモジュール (HSM) を使用して、デバイスのプライベートキーと証明書を安全に保存できます。詳細については、「ハードウェアセキュリティ統合」を参照してください。 バグ修正と機能向上 • nucleus が AWS IoT Core と MQTT 接続を確立している際のランタイム例外に関する問題を修正しました。
2.5.2	バグ修正と機能向上 • Greengrass nucleus の更新後、デバイスを停止または再起動した後、Windows サービスが再び起動しない問題を修正しました。

バージョン	変更
2.5.1	**Warning** このバージョンは現在利用できません。このバージョンの改善は、このコンポーネントのそれ以降のバージョンで利用できます。 バグ修正と機能向上 • Windows の 32 ビットバージョンの Java Runtime Environment (JRE) のサポートが追加されました。 • AWS IoT ポリシーで、`greengrass:ListThingGroupsForCoreDevice` 権限が付与されないコアデバイスのモノグループの削除動作が変更されました。このバージョンでは、デプロイは続行され、警告が記録され、モノグループからコアデバイスを削除してもコンポーネントは削除されません。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。 • Greengrass nucleus が Greengrass コンポーネントプロセスで利用できるようにするシステム環境変数の問題を修正しました。これで、コンポーネントを再起動して、最新のシステム環境変数を使用できます。

バージョン	変更
2.5.0	新機能 • Windows を実行するコアデバイスのサポートが追加されました。 • モノグループを削除する動作が変更されました。このバージョンでは、モノグループからコアデバイスを削除して、次のデプロイでそのモノグループのコンポーネントをアンインストールできるようになりました。 この変更の結果、コアデバイスの AWS IoT ポリシーには アクセス`greengrass:ListThingGroupsForCoreDevice` 許可が必要です。AWS IoT Greengrass Core ソフトウェアインストーラを使用してリソースをプロビジョニングした場合、デフォルトの AWS IoT ポリシーでは`greengrass:*` 、このアクセス許可を含む が許可されます。詳細については、「AWS IoT Greengrass のデバイス認証と認可」を参照してください。 • HTTPS プロキシ設定のサポートが追加されました。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。 • 新しい `windowsUser` 設定パラメータが追加されました。このパラメータを使用して、Windows コアデバイスでコンポーネントを実行するために使用するデフォルトユーザーを指定できます。詳細については、「コンポーネントを実行するユーザーを設定する」を参照してください。 • HTTP リクエストのタイムアウトをカスタマイズして、低速ネットワークでのパフォーマンスを向上させるために使用できる新しい `httpClient` 設定オプションが追加されました。詳細については、httpClient 設定パラメータを参照してください。 バグ修正と機能向上 • コンポーネントからコアデバイスを再起動するためのブートストラップライフサイクルオプションが修正されました。 • recipe 変数でハイフンがサポートされるようになりました。 • オンデマンド Lambda 関数コンポーネントの IPC 認証が修正されました。 • ログメッセージを改善し、クリティカルでないログを `INFO` から `DEBUG` レベルに変更したことで、ログの利便性が高まりました。

バージョン	変更
	• 自動プロビジョニングを使用して AWS IoT Greengrass Core ソフトウェアをインストールするときに Greengrass nucleus が作成するデフォルトのトークン交換ロールから アクセス`iot:DescribeCertificate` 許可を削除します。この権限は Greengrass nucleus では使用されません。 • `iam:CreatePolicy` が、同じポリシーで利用できる場合、自動プロビジョニングスクリプトで `iam:GetPolicy` 権限が必要ないように問題を修正しました。 • 追加のマイナー修正と機能向上。

バージョン	変更
2.4.0	新機能 • システムリソース制限のサポートを追加します。各コンポーネントのプロセスがコアデバイスで使用できる CPU および RAM の最大使用数を設定できます。詳細については、「コンポーネントのシステムリソース制限を設定する」を参照してください。 • コンポーネントを一時停止および再開するための IPC 操作が追加されました。詳細については、PauseComponentおよびResumeComponentを参照してください。 • プラグインのプロビジョニングのサポートが追加されました。インストール時に実行する JAR ファイルを指定して、Greengrass コアデバイスに必要な AWS リソースをプロビジョニングできます。Greengrass nucleus には、カスタムプロビジョニングプラグインを開発するために実装できるインターフェースが含まれています。詳細については、「カスタムリソースプロビジョニングを使用して AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。 • AWS IoT Greengrass Core ソフトウェアインストーラにオプションの `thing-name-policy` 引数を追加します。このオプションを使用すると、自動リソースプロビジョニングを使用して AWS IoT Greengrass Core ソフトウェアをインストールするときに、既存のポリシーまたはカスタム AWS IoT ポリシーを指定できます。 バグ修正と機能向上 • 起動時にログ設定を更新します。これにより、起動時にログ設定が適用されなかった問題が修正されます。 • インストール中に、Greengrass ルートフォルダ内のコンポーネントストアを指すように nucleus ローダーのシンボリックリンクを更新します。この更新により、 AWS IoT Greengrass Core ソフトウェアのインストール時にダウンロードした JAR ファイルやその他の nucleus アーティファクトを削除できます。 • 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。

バージョン	変更
2.3.0	新機能 • 7 KB (モノをターゲットとするデプロイの場合) または 31 KB (モノグループをターゲットとするデプロイの場合) から 10 MB までのデプロイ設定ドキュメントのサポートが追加されました。 この機能を使用するには、コアデバイスの AWS IoT ポリシーで アクセス`greengrass:GetDeploymentConfiguration` 許可を付与する必要があります。AWS IoT Greengrass Core ソフトウェアインストーラを使用してリソースをプロビジョニングした場合、コアデバイスの AWS IoT ポリシーでは`greengrass:*` 、このアクセス許可を含むが許可されます。詳細については、「AWS IoT Greengrass のデバイス認証と認可」を参照してください。 • `iot:thingName` の recipe 変数が追加されました。この recipe 変数を使用して、recipe 内のコアデバイスの AWS IoT モノの名前を取得できます。詳細については、「レシピ変数」を参照してください。 バグ修正と機能向上 • 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。
2.2.0	新機能 • ローカルシャドウ管理用の IPC 操作が追加されました。 バグ修正と機能向上 • JAR ファイルのサイズを小さくします。 • メモリ使用量を削減します。 • 特定のケースでログ設定が更新されなかった問題を修正しました。 • 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。

バージョン	変更

2.1.0	新機能
	• Amazon ECR のプライベートリポジトリからの Docker イメージのダウンロードがサポートされます。
	• 次のパラメータを追加して、コアデバイスの MQTT 設定をカスタマイズします。
	  • `maxInFlightPublishes` - 同時に送信できる未確認 MQTT QoS 1 メッセージの最大数。
	  • `maxPublishRetry` - パブリッシュに失敗したメッセージを再試行する最大回数。
	• `fleetstatusservice` 設定パラメータを追加して、コアデバイスがデバイスステータスを AWS クラウド に公開する間隔を設定します。
	• 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。
	バグ修正と機能向上
	• nucleus が再起動したときにシャドウデプロイが複製される問題を修正しました。
	• サービスロード例外が発生したときに nucleus がクラッシュする問題を修正しました。
	• 循環依存関係を含むデプロイが失敗するように、コンポーネントの依存関係の解決が改善されました。
	• そのコンポーネントがコアデバイスから削除された場合に、プラグインコンポーネントが再デプロイされない問題を修正しました。
	• Lambda コンポーネントや root で実行するコンポーネントの */greengrass/v2* `/work` ディレクトリに `HOME` 環境変数が設定される問題を修正しました。コンポーネントを実行するユーザーのホームディレクトリに `HOME` 変数が正しく設定されるようになりました。
	• 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。

バージョン	変更
2.0.5	バグ修正と機能向上 • AWSが提供するコンポーネントをダウンロードするときに、設定されたネットワークプロキシを介してトラフィックを正しくルーティングします。 • AWS 中国リージョンで正しい Greengrass データプレーンのエンドポイントが使用されます。
2.0.4	新機能 • ポート 443 経由の HTTPS トラフィックが有効になりました。nucleus コンポーネントのバージョン 2.0.4 新しい `greengrassDataPlanePort` 設定パラメータで、デフォルトのポート 8443 ではなくポート 443 を経由するように HTTPS 通信を設定できるようになりました。詳細については、「ポート 443 経由で HTTPS を設定する」を参照してください。 • 作業パスの recipe 変数が追加されました。この recipe 変数を使用して、コンポーネントの作業フォルダへのパスを取得できます。このパスを使用して、コンポーネントとその依存関係間でファイルを共有できます。詳細については、「作業パスの recipe 変数」を参照してください。 バグ修正と機能向上 • ロールポリシーが既に存在する場合、トークン交換 AWS Identity and Access Management (IAM) ロールポリシーの作成を防止します。 この変更の結果、`--provision true` で実行する際にインストーラで `iam:GetPolicy` と `sts:GetCallerIdentity` が必要になります。詳細については、「インストーラがリソースをプロビジョニングするための最小限の IAM ポリシー」を参照してください。 • まだ正常に登録されていないデプロイのキャンセルが正しく処理されるようおになりました。 • デプロイをロールバックするときに、新しいタイムスタンプのある古いエントリが削除されるように設定を更新しました。 • 追加のマイナー修正と機能向上。詳細については、「」のリリースを参照してください GitHub。

バージョン	変更
2.0.3	当初のバージョン

クライアントデバイス認証

クライアントデバイス認証コンポーネント (aws.greengrass.clientdevices.Auth) がクライアントデバイスを認証し、クライアントデバイスのアクションを承認します。

> **Note**
>
> クライアントデバイスは、Greengrass コアデバイスに接続し、処理するために MQTT メッセージとデータを送信するローカル IoT デバイスです。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

トピック

バージョン

> **Note**
>
> クライアントデバイス認証バージョン 2.3.0 は廃止されました。クライアントデバイス認証バージョン 2.3.1 以降にアップグレードすることを強くお勧めします。

このコンポーネントには、次のバージョンがあります。

- 2.4.x
- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

タイプ

このコンポーネントはプラグインコンポーネント (`aws.greengrass.plugin`) です。Greengrass nucleus は、nucleus と同じ Java バーチャルマシン (JVM) でこのコンポーネントを実行します。コアデバイスでこのコンポーネントのバージョンを変更するとき、nucleus が再起動します。

このコンポーネントは、Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Greengrass サービスロールを AWS アカウント に関連づけ、`iot:DescribeCertificate` の権限を許可する必要があります。
- コアデバイスの AWS IoT ポリシーでは以下のアクセス許可を付与する必要があります。
  - `greengrass:GetConnectivityInfo`。リソースには、このコンポーネントを実行するコアデバイスの ARN が含まれます。
  - `greengrass:VerifyClientDeviceIoTCertificateAssociation`。リソースには、コアデバイスに接続する各クライアントデバイスの Amazon リソースネーム (ARN) が含まれます。

  - `greengrass:VerifyClientDeviceIdentity`
  - `greengrass:PutCertificateAuthorities`
  - `iot:Publish`。リソースには、次の MQTT トピックの ARN が含まれます。
    - `$aws/things/coreDeviceThingName*-gci/shadow/get`
  - `iot:Subscribe`。リソースには、次の MQTT トピックフィルターの ARN が含まれます。
    - `$aws/things/coreDeviceThingName*-gci/shadow/update/delta`
    - `$aws/things/coreDeviceThingName*-gci/shadow/get/accepted`
  - `iot:Receive`。リソースには、次の MQTT トピックの ARN が含まれます。
    - `$aws/things/coreDeviceThingName*-gci/shadow/update/delta`
    - `$aws/things/coreDeviceThingName*-gci/shadow/get/accepted`

  詳細については、「データプレーンオペレーションの AWS IoT ポリシー と クライアントデバイスをサポートするための最低限の AWS IoT ポリシー」を参照してください。
- (オプション) オフライン認証を使用するには、AWS IoT Greengrass のサービスによって使用される AWS Identity and Access Management (IAM) ロールに次の許可が含まれている必要があります。
  - コアデバイスでオフライン認証用のクライアントを一覧表示できるようにするための `greengrass:ListClientDevicesAssociatedWithCoreDevice`。
- クライアントデバイス認証コンポーネントは、VPC での実行がサポートされています。このコンポーネントを VPC にデプロイするには、以下が必要です。
  - クライアントデバイス認証コンポーネントには、AWS IoT data、認証情報、および Amazon S3 AWS IoT への接続が必要です。

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
`iot.region.amazonaws.com`	443	はい	AWS IoT モノの証明書に関す

エンドポイント	ポート	必要	説明
			る情報を取得するために使用します。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.4.4

次の表に、このコンポーネントのバージョン 2.4.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.6.0 <2.13.0	ソフト

2.4.3

次の表に、このコンポーネントのバージョン 2.4.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.6.0 <2.12.0	ソフト

2.4.1 and 2.4.2

次の表に、このコンポーネントのバージョン 2.4.1 および 2.4.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.6.0 <2.11.0	ソフト

2.3.0 – 2.4.0

次の表に、このコンポーネントのバージョン 2.3.0 から 2.4.0 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.6.0 <2.10.0	ソフト

2.3.0

次の表に、このコンポーネントのバージョン 2.3.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.6.0 <2.10.0	ソフト

2.2.3

次の表に、このコンポーネントのバージョン 2.2.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.6.0 <=2.9.0	ソフト

2.2.2

次の表に、このコンポーネントのバージョン 2.2.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.6.0 <=2.8.0	ソフト

2.2.1

次の表に、このコンポーネントのバージョン 2.2.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.6.0 <2.8.0	ソフト

2.2.0

次の表に、このコンポーネントのバージョン 2.2.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.6.0 <2.7.0	ソフト

2.1.0

次の表に、このコンポーネントのバージョン 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.7.0	ソフト

2.0.4

次の表に、このコンポーネントのバージョン 2.0.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.6.0	ソフト

2.0.2 and 2.0.3

次の表に、このコンポーネントのバージョン 2.0.2 および 2.0.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.5.0	ソフト

2.0.1

次の表に、このコンポーネントのバージョン 2.0.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.4.0	ソフト

2.0.0

次の表に、このコンポーネントのバージョン 2.0.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.3.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

v2.4.5

`deviceGroups`

デバイスグループは、コアデバイスに接続して通信する権限を持つクライアントデバイスのグループです。選択ルールを使用してクライアントデバイスのグループを特定し、各デバイスグループにアクセス許可を指定する、クライアントデバイス認証ポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

`formatVersion`

この設定オブジェクトの形式バージョン。

次のオプションから選択します。

- `2021-03-05`

`definitions`

このコアデバイスのデバイスグループ。各定義では、クライアントデバイスがグループのメンバーであるかどうかを評価する選択ルールを指定します。各定義では、選択ルールに一致するクライアントデバイスに適用するアクセス権限ポリシーも指定します。クライアントデバイスが複数のデバイスグループのメンバーである場合、デバイスのアクセス許可は各グループのアクセス許可ポリシーで設定されます。

このオブジェクトには、次の情報が含まれます。

`groupNameKey`

このデバイスグループの名前。を、このデバイスグループを識別するのに役立つ名前*`groupNameKey`*に置き換えます。

このオブジェクトには、次の情報が含まれます。

`selectionRule`

このデバイスグループのメンバーであるクライアントデバイスを指定するクエリ。クライアントデバイスが接続されると、コアデバイスはこの選択ルールを評価して、クライアントデバイスがこのデバイスグループのメンバーであるかどうかを判断します。クライアントデバイスがメンバーである場合、コアデバイスはこのデバイスグループのポリシーを使用して、クライアントデバイスのアクションを承認します。

各選択ルールは少なくとも 1 つの選択ルール句で設定されています。この句は、クライアントデバイスと一致させることができる 1 つの式クエリです。選択ルールは、AWS IoT フリートのインデックス作成と同じクエリ構文を使用します。選択ルールの構文に関する詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT フリートのインデックス作成クエリ構文」を参照してください。

* ワイルドカードを使用して、複数のクライアントデバイスを 1 つの選択ルール句と一致させます。モノの名前の先頭と末尾にこのワイルドカードを使用して、指定

した文字列で始まる、または終わる名前のクライアントデバイスと一致させることができます。このワイルドカードを使用して、すべてのクライアントデバイスと一致させることもできます。

> **Note**
>
> コロン文字 (:) を含む値を選択する場合は、コロン文字にバックスラッシュ文字 (\) を使用してエスケープします。JSON などの形式では、バックスラッシュ文字をエスケープする必要があるため、コロン文字の前に 2 つのバックスラッシュ文字を入力する必要があります。例えば、`thingName: MyTeam\\:ClientDevice1` が `MyTeam:ClientDevice1` という名前のモノを選択するように指定する場合です。

次のセレクターを指定できます。

- `thingName` - クライアントデバイスの AWS IoT モノの名前。

Example 選択ルール例

次の選択ルールは、名前が `MyClientDevice1` または `MyClientDevice2` のクライアントデバイスと一致します。

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example 選択ルール例 (ワイルドカードを使用)

次の選択ルールは、名前が `MyClientDevice` で始まるクライアントデバイスと一致します。

```
thingName: MyClientDevice*
```

Example 選択ルール例 (ワイルドカードを使用)

次の選択ルールは、名前が で終わるクライアントデバイスと一致しますMyClientDevice。

```
thingName: *MyClientDevice
```

Example 選択ルール例 (すべてのデバイスに一致)

次の選択ルールは、すべてのクライアントデバイスと一致します。

```
thingName: *
```

`policyName`

このデバイスグループ内のクライアントデバイスに適用されるアクセス許可ポリシー。`policies` オブジェクトで定義するポリシーの名前を指定します。

`policies`

コアデバイスに接続するクライアントデバイス用の、クライアントデバイス認証ポリシー。各承認ポリシーは、クライアントデバイスがこれらのアクションを実行できるアクションセットとリソースを指定しています。

このオブジェクトには、次の情報が含まれます。

`policyNameKey`

この承認ポリシーの名前。を、この承認ポリシーを識別するのに役立つ名前*`policyNameKey`*に置き換えます。このポリシー名を使用して、デバイスグループに適用するポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

`statementNameKey`

このポリシーステートメントの名前。を、このポリシーステートメントを識別するのに役立つ名前*`statementNameKey`*に置き換えます。

このオブジェクトには、次の情報が含まれます。

`operations`

このポリシーのリソースに許可する操作のリスト。

以下のすべての操作を含めることができます。

- `mqtt:connect` - コアデバイスに接続するアクセス許可を付与します。コアデバイスに接続するには、クライアントデバイスにこのアクセス許可が必要です。

この操作は、次のリソースをサポートします。

- `mqtt:clientId:`*`deviceClientId`* - クライアントデバイスがコアデバイスの MQTT ブローカーへの接続に使用するクライアント ID に基づいて、アクセスを制限します。を使用するクライアント ID *`deviceClientId`*に置き換えます。

- `mqtt:publish` - MQTT メッセージをトピックにパブリッシュする権限を付与します。

  この操作は、次のリソースをサポートします。

  - `mqtt:topic:`*`mqttTopic`* - クライアントデバイスがメッセージをパブリッシュする MQTT トピックに基づいて、アクセスを制限します。*`mqttTopic`* を使用するトピックに置き換えます。

    このリソースは MQTT トピックのワイルドカードをサポートしていません。

- `mqtt:subscribe` - メッセージを受信する MQTT トピックフィルターにサブスクライブする許可を付与します。

  この操作は、次のリソースをサポートします。

  - `mqtt:topicfilter:`*`mqttTopicFilter`* - クライアントデバイスがメッセージにサブスクライブできる MQTT トピックに基づいて、アクセスを制限します。を使用するトピックフィルター*`mqttTopicFilter`*に置き換えます。

    このリソースは + および # MQTT トピックのワイルドカードをサポートしています。詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT トピック」を参照してください。

    クライアントデバイスは、許可を付与したトピックフィルターにのみサブスクライブできます。例えば、クライアントデバイスに `mqtt:topicfilter:client/+/status` リソースへのサブスクライブを許可した場合、クライアントデバイスは `client/+/status` にサブスクライブできますが、`client/client1/status` にはサブスクライブできません。

* ワイルドカードを指定すれば、すべてのアクションへのアクセスを許可することができます。

`resources`

このポリシーでの操作を許可するリソースのリスト。このポリシーでの操作に対応するリソースを指定します。例えば、`mqtt:publish` 操作を指定するポリシーで、MQTT トピックリソース (`mqtt:topic:`*`mqttTopic`*) のリストを指定することができます。

`*` ワイルドカードを指定すれば、すべてのリソースへのアクセスを許可することができます。`*` ワイルドカードを使用して、部分的なリソース識別子と一致させることはできません。例えば、**`"resources": "*"`** と指定することはできますが、**`"resources": "mqtt:clientId:*"`** と指定することはできません。

`statementDescription`

(オプション) このポリシーステートメントの説明。

`certificates`

(オプション) このコアデバイスの証明書設定オプション。このオブジェクトには、次の情報が含まれます。

`serverCertificateValiditySeconds`

(オプション) ローカル MQTT サーバー証明書の有効期限が切れるまでの時間 (秒単位)。このオプションを設定すれば、クライアントデバイスがコアデバイスから切断し、また再接続する回数をカスタマイズすることができます。

このコンポーネントは、有効期限が切れる 24 時間前に、ローカル MQTT サーバー証明書をローテーションします。Moquette MQTT ブローカーコンポーネントなどの MQTT ブローカーで、新しい証明書を生成して再起動します。これが実行されると、このコアデバイスに接続されているすべてのクライアントデバイスが切断されます。クライアントデバイスは、短時間待機した後に、コアデバイスに再度接続できます。

デフォルト: `604800` (7 日)

最小値: `172800` (2 日)

最大値: `864000` (10 日)

`performance`

(オプション) このコアデバイスのパフォーマンス設定オプション。このオブジェクトには、次の情報が含まれます。

`maxActiveAuthTokens`

(オプション) アクティブなクライアントデバイス認証トークンの最大数。この数を増やすと、より多くのクライアントデバイスが、再認証せずに単一のコアデバイスに接続できるようになります。

デフォルト: `2500`

`cloudRequestQueueSize`

(オプション) このコンポーネントで拒否される前に、キューに格納される AWS クラウドリクエストの最大数。

デフォルト: `100`

`maxConcurrentCloudRequests`

(オプション) AWS クラウド に同時に送信できるリクエストの最大数。この数を増やすと、多数のクライアントデバイスを接続するコアデバイスで、認証のパフォーマンスを向上させることができます。

デフォルト: `1`

`certificateAuthority`

(オプション) コアデバイスの中間認証機関を独自の中間認証機関に置き換える認証機関設定オプション。

> **Note**
>
> Greengrass コアデバイスをカスタム認証機関 (CA) で設定し、同じ CA を使用してクライアントデバイス証明書を発行する場合、Greengrass はクライアントデバイス MQTT オペレーションの承認ポリシーチェックをバイパスします。クライアントデバイス認証コンポーネントは、使用するように設定された CA によって署名された証明書を使用してクライアントを完全に信頼します。
> カスタム CA を使用するときにこの動作を制限するには、別の CA または中間 CA を使用してクライアントデバイスを作成して署名し、正しい中間 CA を指すように `certificateUri`および `certificateChainUri`フィールドを調整します。

このオブジェクトには、次の情報が含まれます。

certificateUri

証明書の位置。ファイルシステム URI、またはハードウェアセキュリティモジュールに保存されている証明書をポイントする URI にすることができます。

certificateChainUri

コアデバイス CA の証明書チェーンの場所。これは、ルート CA に対する完全な証明書チェーンとなります。ファイルシステム URI、またはハードウェアセキュリティモジュールに保存されている証明書チェーンをポイントする URI にすることができます。

privateKeyUri

コアデバイスのプライベートキーの場所。これは、ファイルシステム URI、またはハードウェアセキュリティモジュールに保存されている証明書プライベートキーをポイントする URI にすることができます。

security

(オプション) このコアデバイスのセキュリティ設定オプション。このオブジェクトには、次の情報が含まれます。

clientDeviceTrustDurationMinutes

クライアントデバイスの認証情報を信頼できる時間 (分)。この時間が経過すると、コアデバイスでの再認証が必要となります。デフォルト値は 1 です。

metrics

(オプション) このコアデバイスのメトリックオプション。エラーメトリックは、クライアントデバイスの認証にエラーがある場合にのみ表示されます。このオブジェクトには、次の情報が含まれます。

disableMetrics

disableMetrics フィールドが true に設定されている場合、クライアントデバイス認証はメトリックを収集しません。

デフォルト: false

aggregatePeriodSeconds

クライアントデバイス認証がメトリクスを集約してテレメトリエージェントに送信する頻度を決定する集計期間 (秒単位)。テレメトリエージェントは依然として 1 日に 1 回メトリクスを公開するので、メトリクスの公開頻度は変わりません。

デフォルト: 3600

startupTimeoutSeconds

(オプション) コンポーネントが起動する最大時間 (秒)。このタイムアウトを超えている場合、コンポーネントの状態が BROKEN に変わります。

デフォルト: 120

Example 例: 設定マージの更新 (制限ポリシーを使用)

次の設定例では、名前が MyClientDevice で始まるクライアントデバイスに対して、すべてのトピックの接続とパブリッシュ/サブスクライブを許可するように指定しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
```

```
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example 例: 設定マージの更新 (許容ポリシーを使用)

次の設定例では、すべてのクライアントデバイスに対して、すべてのトピックに接続しパブリッシュ/サブスクライブすることを許可しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
      }
    },
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
```

```
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

v2.4.2 - v2.4.4

`deviceGroups`

デバイスグループは、コアデバイスに接続して通信する権限を持つクライアントデバイスのグループです。選択ルールを使用してクライアントデバイスのグループを特定し、各デバイスグループにアクセス許可を指定する、クライアントデバイス認証ポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

`formatVersion`

この設定オブジェクトの形式バージョン。

次のオプションから選択します。

- `2021-03-05`

`definitions`

このコアデバイスのデバイスグループ。各定義では、クライアントデバイスがグループのメンバーであるかどうかを評価する選択ルールを指定します。各定義では、選択ルールに一致するクライアントデバイスに適用するアクセス権限ポリシーも指定します。クライアントデバイスが複数のデバイスグループのメンバーである場合、デバイスのアクセス許可は各グループのアクセス許可ポリシーで設定されます。

このオブジェクトには、次の情報が含まれます。

`groupNameKey`

このデバイスグループの名前。を、このデバイスグループを識別するのに役立つ名前*`groupNameKey`*に置き換えます。

このオブジェクトには、次の情報が含まれます。

`selectionRule`

このデバイスグループのメンバーであるクライアントデバイスを指定するクエリ。クライアントデバイスが接続されると、コアデバイスはこの選択ルールを評価して、クライアントデバイスがこのデバイスグループのメンバーであるかどうかを判断します。クライアントデバイスがメンバーである場合、コアデバイスはこのデバイスグループのポリシーを使用して、クライアントデバイスのアクションを承認します。

各選択ルールは少なくとも 1 つの選択ルール句で設定されています。この句は、クライアントデバイスと一致させることができる 1 つの式クエリです。選択ルールは、AWS IoT フリートのインデックス作成と同じクエリ構文を使用します。選択ルールの構文に関する詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT フリートのインデックス作成クエリ構文」を参照してください。

* ワイルドカードを使用して、複数のクライアントデバイスを 1 つの選択ルール句と一致させます。モノ名の末尾にこのワイルドカードを使用すると、指定した文字列で始まる名前のクライアントデバイスと一致させることができます。このワイルドカードを使用して、すべてのクライアントデバイスと一致させることもできます。

> **Note**
>
> コロン文字 (`:`) を含む値を選択する場合は、コロン文字にバックスラッシュ文字 (`\\`) を使用してエスケープします。JSON などの形式では、バックスラッシュ文字をエスケープする必要があるため、コロン文字の前に 2 つのバックスラッシュ文字を入力する必要があります。例えば、`thingName:MyTeam\\\\:ClientDevice1` が `MyTeam:ClientDevice1` という名前のモノを選択するように指定する場合です。

次のセレクターを指定できます。

- `thingName` - クライアントデバイスの AWS IoT モノの名前。

Example 選択ルール例

次の選択ルールは、名前が `MyClientDevice1` または `MyClientDevice2` のクライアントデバイスと一致します。

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example 選択ルール例 (ワイルドカードを使用)

次の選択ルールは、名前が MyClientDevice で始まるクライアントデバイスと一致します。

```
thingName: MyClientDevice*
```

Example 選択ルール例 (すべてのデバイスに一致)

次の選択ルールは、すべてのクライアントデバイスと一致します。

```
thingName: *
```

`policyName`

このデバイスグループ内のクライアントデバイスに適用されるアクセス許可ポリシー。policies オブジェクトで定義するポリシーの名前を指定します。

`policies`

コアデバイスに接続するクライアントデバイス用の、クライアントデバイス認証ポリシー。各承認ポリシーは、クライアントデバイスがこれらのアクションを実行できるアクションセットとリソースを指定しています。

このオブジェクトには、次の情報が含まれます。

`policyNameKey`

この承認ポリシーの名前。を、この承認ポリシーを識別するのに役立つ名前*`policyNameKey`*に置き換えます。このポリシー名を使用して、デバイスグループに適用するポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

`statementNameKey`

このポリシーステートメントの名前。を、このポリシーステートメントを識別するのに役立つ名前*`statementNameKey`*に置き換えます。

このオブジェクトには、次の情報が含まれます。

`operations`

このポリシーのリソースに許可する操作のリスト。

以下のすべての操作を含めることができます。

- `mqtt:connect` - コアデバイスに接続するアクセス許可を付与します。コアデバイスに接続するには、クライアントデバイスにこのアクセス許可が必要です。

  この操作は、次のリソースをサポートします。

  - `mqtt:clientId:`*`deviceClientId`* - クライアントデバイスがコアデバイスの MQTT ブローカーへの接続に使用するクライアント ID に基づいて、アクセスを制限します。を使用するクライアント ID *`deviceClientId`*に置き換えます。

- `mqtt:publish` - MQTT メッセージをトピックにパブリッシュする権限を付与します。

  この操作は、次のリソースをサポートします。

  - `mqtt:topic:`*`mqttTopic`* - クライアントデバイスがメッセージをパブリッシュする MQTT トピックに基づいて、アクセスを制限します。*`mqttTopic`* を使用するトピックに置き換えます。

    このリソースは MQTT トピックのワイルドカードをサポートしていません。

- `mqtt:subscribe` - メッセージを受信する MQTT トピックフィルターにサブスクライブする許可を付与します。

  この操作は、次のリソースをサポートします。

  - `mqtt:topicfilter:`*`mqttTopicFilter`* - クライアントデバイスがメッセージにサブスクライブできる MQTT トピックに基づいて、アクセスを制限します。を使用するトピックフィルター*`mqttTopicFilter`*に置き換えます。

    このリソースは + および # MQTT トピックのワイルドカードをサポートしています。詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT トピック」を参照してください。

クライアントデバイスは、許可を付与したトピックフィルターにのみサブスクライブできます。例えば、クライアントデバイスに mqtt:topicfilter:client/+/status リソースへのサブスクライブを許可した場合、クライアントデバイスは client/+/status にサブスクライブできますが、client/client1/status にはサブスクライブできません。

* ワイルドカードを指定すれば、すべてのアクションへのアクセスを許可することができます。

resources

このポリシーでの操作を許可するリソースのリスト。このポリシーでの操作に対応するリソースを指定します。例えば、mqtt:publish 操作を指定するポリシーで、MQTT トピックリソース (mqtt:topic:*mqttTopic*) のリストを指定することができます。

* ワイルドカードを指定すれば、すべてのリソースへのアクセスを許可することができます。* ワイルドカードを使用して、部分的なリソース識別子と一致させることはできません。例えば、**"resources": "*"** と指定することはできますが、**"resources": "mqtt:clientId:*"** と指定することはできません。

statementDescription

(オプション) このポリシーステートメントの説明。

certificates

(オプション) このコアデバイスの証明書設定オプション。このオブジェクトには、次の情報が含まれます。

serverCertificateValiditySeconds

(オプション) ローカル MQTT サーバー証明書の有効期限が切れるまでの時間 (秒単位)。このオプションを設定すれば、クライアントデバイスがコアデバイスから切断し、また再接続する回数をカスタマイズすることができます。

このコンポーネントは、有効期限が切れる 24 時間前に、ローカル MQTT サーバー証明書をローテーションします。Moquette MQTT ブローカーコンポーネントなどの MQTT ブローカーで、新しい証明書を生成して再起動します。これが実行されると、このコアデバイスに接続されているすべてのクライアントデバイスが切断されます。クライアントデバイスは、短時間待機した後に、コアデバイスに再度接続できます。

デフォルト: 604800 (7 日)

最小値: 172800 (2 日)

最大値: 864000 (10 日)

`performance`

(オプション) このコアデバイスのパフォーマンス設定オプション。このオブジェクトには、次の情報が含まれます。

`maxActiveAuthTokens`

(オプション) アクティブなクライアントデバイス認証トークンの最大数。この数を増やすと、より多くのクライアントデバイスが、再認証せずに単一のコアデバイスに接続できるようになります。

デフォルト: 2500

`cloudRequestQueueSize`

(オプション) このコンポーネントで拒否される前に、キューに格納される AWS クラウドリクエストの最大数。

デフォルト: 100

`maxConcurrentCloudRequests`

(オプション) AWS クラウド に同時に送信できるリクエストの最大数。この数を増やすと、多数のクライアントデバイスを接続するコアデバイスで、認証のパフォーマンスを向上させることができます。

デフォルト: 1

`certificateAuthority`

(オプション) コアデバイスの中間認証機関を独自の中間認証機関に置き換える認証機関設定オプション。

> **Note**
>
> Greengrass コアデバイスをカスタム認証機関 (CA) で設定し、同じ CA を使用してクライアントデバイス証明書を発行する場合、Greengrass はクライアントデバイス MQTT オペレーションの承認ポリシーチェックをバイパスします。クライアントデバ

イス認証コンポーネントは、使用するように設定された CA によって署名された証明書を使用してクライアントを完全に信頼します。
カスタム CA を使用するときにこの動作を制限するには、別の CA または中間 CA を使用してクライアントデバイスを作成して署名し、正しい中間 CA を指すように `certificateUri`および `certificateChainUri`フィールドを調整します。

このオブジェクトには、次の情報が含まれます。

certificateUri

証明書の位置。ファイルシステム URI、またはハードウェアセキュリティモジュールに保存されている証明書をポイントする URI にすることができます。

`certificateChainUri`

コアデバイス CA の証明書チェーンの場所。これは、ルート CA に対する完全な証明書チェーンとなります。ファイルシステム URI、またはハードウェアセキュリティモジュールに保存されている証明書チェーンをポイントする URI にすることができます。

`privateKeyUri`

コアデバイスのプライベートキーの場所。これは、ファイルシステム URI、またはハードウェアセキュリティモジュールに保存されている証明書プライベートキーをポイントする URI にすることができます。

`security`

(オプション) このコアデバイスのセキュリティ設定オプション。このオブジェクトには、次の情報が含まれます。

`clientDeviceTrustDurationMinutes`

クライアントデバイスの認証情報を信頼できる時間 (分)。この時間が経過すると、コアデバイスでの再認証が必要となります。デフォルト値は 1 です。

`metrics`

(オプション) このコアデバイスのメトリックオプション。エラーメトリックは、クライアントデバイスの認証にエラーがある場合にのみ表示されます。このオブジェクトには、次の情報が含まれます。

disableMetrics

disableMetrics フィールドが true に設定されている場合、クライアントデバイス認証はメトリックを収集しません。

デフォルト: false

aggregatePeriodSeconds

クライアントデバイス認証がメトリクスを集約してテレメトリエージェントに送信する頻度を決定する集計期間 (秒単位)。テレメトリエージェントは依然として 1 日に 1 回メトリクスを公開するので、メトリクスの公開頻度は変わりません。

デフォルト: 3600

startupTimeoutSeconds

(オプション) コンポーネントが起動する最大時間 (秒)。このタイムアウトを超えている場合、コンポーネントの状態が BROKEN に変わります。

デフォルト: 120

Example 例: 設定マージの更新 (制限ポリシーを使用)

次の設定例では、名前が MyClientDevice で始まるクライアントデバイスに対して、すべてのトピックの接続とパブリッシュ/サブスクライブを許可するように指定しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
```

```
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example 例: 設定マージの更新 (許容ポリシーを使用)

次の設定例では、すべてのクライアントデバイスに対して、すべてのトピックに接続しパブリッシュ/サブスクライブすることを許可しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
      }
    },
```

```
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

v2.4.0 - v2.4.1

`deviceGroups`

デバイスグループは、コアデバイスに接続して通信する権限を持つクライアントデバイスのグループです。選択ルールを使用してクライアントデバイスのグループを特定し、各デバイスグループにアクセス許可を指定する、クライアントデバイス認証ポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

`formatVersion`

この設定オブジェクトの形式バージョン。

次のオプションから選択します。

- `2021-03-05`

`definitions`

このコアデバイスのデバイスグループ。各定義では、クライアントデバイスがグループのメンバーであるかどうかを評価する選択ルールを指定します。各定義では、選択ルールに一致するクライアントデバイスに適用するアクセス権限ポリシーも指定します。クライアントデバイスが複数のデバイスグループのメンバーである場合、デバイスのアクセス許可は各グループのアクセス許可ポリシーで設定されます。

このオブジェクトには、次の情報が含まれます。

groupNameKey

このデバイスグループの名前。を、このデバイスグループを識別するのに役立つ名前*groupNameKey*に置き換えます。

このオブジェクトには、次の情報が含まれます。

`selectionRule`

このデバイスグループのメンバーであるクライアントデバイスを指定するクエリ。クライアントデバイスが接続されると、コアデバイスはこの選択ルールを評価して、クライアントデバイスがこのデバイスグループのメンバーであるかどうかを判断します。クライアントデバイスがメンバーである場合、コアデバイスはこのデバイスグループのポリシーを使用して、クライアントデバイスのアクションを承認します。

各選択ルールは少なくとも 1 つの選択ルール句で設定されています。この句は、クライアントデバイスと一致させることができる 1 つの式クエリです。選択ルールは、AWS IoT フリートのインデックス作成と同じクエリ構文を使用します。選択ルールの構文に関する詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT フリートのインデックス作成クエリ構文」を参照してください。

* ワイルドカードを使用して、複数のクライアントデバイスを 1 つの選択ルール句と一致させます。モノ名の末尾にこのワイルドカードを使用すると、指定した文字列で始まる名前のクライアントデバイスと一致させることができます。このワイルドカードを使用して、すべてのクライアントデバイスと一致させることもできます。

> **Note**
>
> コロン文字 (:) を含む値を選択する場合は、コロン文字にバックスラッシュ文字 (\\) を使用してエスケープします。JSON などの形式では、バックスラッシュ文字をエスケープする必要があるため、コロン文字の前に 2 つのバックスラッシュ文字を入力する必要があります。例えば、`thingName:MyTeam\\\\:ClientDevice1` が `MyTeam:ClientDevice1` という名前のモノを選択するように指定する場合です。

次のセレクターを指定できます。

- `thingName` - クライアントデバイスの AWS IoT モノの名前。

Example 選択ルール例

次の選択ルールは、名前が `MyClientDevice1` または `MyClientDevice2` のクライアントデバイスと一致します。

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example 選択ルール例 (ワイルドカードを使用)

次の選択ルールは、名前が `MyClientDevice` で始まるクライアントデバイスと一致します。

```
thingName: MyClientDevice*
```

Example 選択ルール例 (すべてのデバイスに一致)

次の選択ルールは、すべてのクライアントデバイスと一致します。

```
thingName: *
```

`policyName`

このデバイスグループ内のクライアントデバイスに適用されるアクセス許可ポリシー。`policies` オブジェクトで定義するポリシーの名前を指定します。

`policies`

コアデバイスに接続するクライアントデバイス用の、クライアントデバイス認証ポリシー。各承認ポリシーは、クライアントデバイスがこれらのアクションを実行できるアクションセットとリソースを指定しています。

このオブジェクトには、次の情報が含まれます。

`policyNameKey`

この承認ポリシーの名前。を、この承認ポリシーを識別するのに役立つ名前*`policyNameKey`*に置き換えます。このポリシー名を使用して、デバイスグループに適用するポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

statementNameKey

このポリシーステートメントの名前。を、このポリシーステートメントを識別するのに役立つ名前*statementNameKey*に置き換えます。

このオブジェクトには、次の情報が含まれます。

operations

このポリシーのリソースに許可する操作のリスト。

以下のすべての操作を含めることができます。

- mqtt:connect - コアデバイスに接続するアクセス許可を付与します。コアデバイスに接続するには、クライアントデバイスにこのアクセス許可が必要です。

  この操作は、次のリソースをサポートします。

  - mqtt:clientId:*deviceClientId* - クライアントデバイスがコアデバイスの MQTT ブローカーへの接続に使用するクライアント ID に基づいて、アクセスを制限します。を使用するクライアント ID *deviceClientId*に置き換えます。
- mqtt:publish - MQTT メッセージをトピックにパブリッシュする権限を付与します。

  この操作は、次のリソースをサポートします。

  - mqtt:topic:*mqttTopic* - クライアントデバイスがメッセージをパブリッシュする MQTT トピックに基づいて、アクセスを制限します。*mqttTopic* を使用するトピックに置き換えます。

    このリソースは MQTT トピックのワイルドカードをサポートしていません。
- mqtt:subscribe - メッセージを受信する MQTT トピックフィルターにサブスクライブする許可を付与します。

  この操作は、次のリソースをサポートします。

  - mqtt:topicfilter:*mqttTopicFilter* - クライアントデバイスがメッセージにサブスクライブできる MQTT トピックに基づいて、アクセスを制

限します。を使用するトピックフィルター*mqttTopicFilter*に置き換えます。

このリソースは + および # MQTT トピックのワイルドカードをサポートしています。詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT トピック」を参照してください。

クライアントデバイスは、許可を付与したトピックフィルターにのみサブスクライブできます。例えば、クライアントデバイスに `mqtt:topicfilter:client/+/status` リソースへのサブスクライブを許可した場合、クライアントデバイスは `client/+/status` にサブスクライブできますが、`client/client1/status` にはサブスクライブできません。

* ワイルドカードを指定すれば、すべてのアクションへのアクセスを許可することができます。

`resources`

このポリシーでの操作を許可するリソースのリスト。このポリシーでの操作に対応するリソースを指定します。例えば、`mqtt:publish` 操作を指定するポリシーで、MQTT トピックリソース (`mqtt:topic:`*mqttTopic*) のリストを指定することができます。

* ワイルドカードを指定すれば、すべてのリソースへのアクセスを許可することができます。* ワイルドカードを使用して、部分的なリソース識別子と一致させることはできません。例えば、**`"resources": "*"`** と指定することはできますが、**`"resources": "mqtt:clientId:*"`** と指定することはできません。

`statementDescription`

(オプション) このポリシーステートメントの説明。

`certificates`

(オプション) このコアデバイスの証明書設定オプション。このオブジェクトには、次の情報が含まれます。

`serverCertificateValiditySeconds`

(オプション) ローカル MQTT サーバー証明書の有効期限が切れるまでの時間 (秒単位)。このオプションを設定すれば、クライアントデバイスがコアデバイスから切断し、また再接続する回数をカスタマイズすることができます。

このコンポーネントは、有効期限が切れる 24 時間前に、ローカル MQTT サーバー証明書をローテーションします。Moquette MQTT ブローカーコンポーネントなどの MQTT ブローカーで、新しい証明書を生成して再起動します。これが実行されると、このコアデバイスに接続されているすべてのクライアントデバイスが切断されます。クライアントデバイスは、短時間待機した後に、コアデバイスに再度接続できます。

デフォルト: `604800` (7 日)

最小値: `172800` (2 日)

最大値: `864000` (10 日)

`performance`

(オプション) このコアデバイスのパフォーマンス設定オプション。このオブジェクトには、次の情報が含まれます。

`maxActiveAuthTokens`

(オプション) アクティブなクライアントデバイス認証トークンの最大数。この数を増やすと、より多くのクライアントデバイスが、再認証せずに単一のコアデバイスに接続できるようになります。

デフォルト: `2500`

`cloudRequestQueueSize`

(オプション) このコンポーネントで拒否される前に、キューに格納される AWS クラウドリクエストの最大数。

デフォルト: `100`

`maxConcurrentCloudRequests`

(オプション) AWS クラウド に同時に送信できるリクエストの最大数。この数を増やすと、多数のクライアントデバイスを接続するコアデバイスで、認証のパフォーマンスを向上させることができます。

デフォルト: `1`

`certificateAuthority`

(オプション) コアデバイスの中間認証機関を独自の中間認証機関に置き換える認証機関設定オプション。このオブジェクトには、次の情報が含まれます。

このオブジェクトには、次の情報が含まれます。

certificateUri

証明書の位置。ファイルシステム URI、またはハードウェアセキュリティモジュールに保存されている証明書をポイントする URI にすることができます。

certificateChainUri

コアデバイス CA の証明書チェーンの場所。これは、ルート CA に対する完全な証明書チェーンとなります。ファイルシステム URI、またはハードウェアセキュリティモジュールに保存されている証明書チェーンをポイントする URI にすることができます。

privateKeyUri

コアデバイスのプライベートキーの場所。これは、ファイルシステム URI、またはハードウェアセキュリティモジュールに保存されている証明書プライベートキーをポイントする URI にすることができます。

security

(オプション) このコアデバイスのセキュリティ設定オプション。このオブジェクトには、次の情報が含まれます。

clientDeviceTrustDurationMinutes

クライアントデバイスの認証情報を信頼できる時間 (分)。この時間が経過すると、コアデバイスでの再認証が必要となります。デフォルト値は 1 です。

metrics

(オプション) このコアデバイスのメトリックオプション。エラーメトリックは、クライアントデバイスの認証にエラーがある場合にのみ表示されます。このオブジェクトには、次の情報が含まれます。

disableMetrics

disableMetrics フィールドが true に設定されている場合、クライアントデバイス認証はメトリックを収集しません。

デフォルト: false

`aggregatePeriodSeconds`

クライアントデバイス認証がメトリクスを集約してテレメトリエージェントに送信する頻度を決定する集計期間 (秒単位)。テレメトリエージェントは依然として 1 日に 1 回メトリクスを公開するので、メトリクスの公開頻度は変わりません。

デフォルト: `3600`

Example 例: 設定マージの更新 (制限ポリシーを使用)

次の設定例では、名前が `MyClientDevice` で始まるクライアントデバイスに対して、すべてのトピックの接続とパブリッシュ/サブスクライブを許可するように指定しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
```

```
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example 例: 設定マージの更新 (許容ポリシーを使用)

次の設定例では、すべてのクライアントデバイスに対して、すべてのトピックに接続しパブリッシュ/サブスクライブすることを許可しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
      }
    },
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

v2.3.x

`deviceGroups`

デバイスグループは、コアデバイスに接続して通信する権限を持つクライアントデバイスのグループです。選択ルールを使用してクライアントデバイスのグループを特定し、各デバイスグループにアクセス許可を指定する、クライアントデバイス認証ポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

`formatVersion`

この設定オブジェクトの形式バージョン。

次のオプションから選択します。

- `2021-03-05`

`definitions`

このコアデバイスのデバイスグループ。各定義では、クライアントデバイスがグループのメンバーであるかどうかを評価する選択ルールを指定します。各定義では、選択ルールに一致するクライアントデバイスに適用するアクセス権限ポリシーも指定します。クライアントデバイスが複数のデバイスグループのメンバーである場合、デバイスのアクセス許可は各グループのアクセス許可ポリシーで設定されます。

このオブジェクトには、次の情報が含まれます。

`groupNameKey`

このデバイスグループの名前。を、このデバイスグループを識別するのに役立つ名前*`groupNameKey`*に置き換えます。

このオブジェクトには、次の情報が含まれます。

`selectionRule`

このデバイスグループのメンバーであるクライアントデバイスを指定するクエリ。クライアントデバイスが接続されると、コアデバイスはこの選択ルールを評価して、クライアントデバイスがこのデバイスグループのメンバーであるかどうかを判断します。クライアントデバイスがメンバーである場合、コアデバイスはこのデバイスグループのポリシーを使用して、クライアントデバイスのアクションを承認します。

各選択ルールは少なくとも 1 つの選択ルール句で設定されています。この句は、クライアントデバイスと一致させることができる 1 つの式クエリです。選択ルールは、AWS IoT フリートのインデックス作成と同じクエリ構文を使用します。選択ルールの構文に関する詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT フリートのインデックス作成クエリ構文」を参照してください。

* ワイルドカードを使用して、複数のクライアントデバイスを 1 つの選択ルール句と一致させます。モノ名の末尾にこのワイルドカードを使用すると、指定した文字列で始まる名前のクライアントデバイスと一致させることができます。このワイルドカードを使用して、すべてのクライアントデバイスと一致させることもできます。

> **Note**
>
> コロン文字 (:) を含む値を選択する場合は、コロン文字にバックスラッシュ文字 (\\) を使用してエスケープします。JSON などの形式では、バックスラッシュ文字をエスケープする必要があるため、コロン文字の前に 2 つのバックスラッシュ文字を入力する必要があります。例えば、`thingName:MyTeam\\\\:ClientDevice1` が `MyTeam:ClientDevice1` という名前のモノを選択するように指定する場合です。

次のセレクターを指定できます。

- `thingName` - クライアントデバイスの AWS IoT モノの名前。

Example 選択ルール例

次の選択ルールは、名前が `MyClientDevice1` または `MyClientDevice2` のクライアントデバイスと一致します。

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example 選択ルール例 (ワイルドカードを使用)

次の選択ルールは、名前が `MyClientDevice` で始まるクライアントデバイスと一致します。

```
thingName: MyClientDevice*
```

Example 選択ルール例 (すべてのデバイスに一致)

次の選択ルールは、すべてのクライアントデバイスと一致します。

```
thingName: *
```

`policyName`

このデバイスグループ内のクライアントデバイスに適用されるアクセス許可ポリシー。`policies` オブジェクトで定義するポリシーの名前を指定します。

`policies`

コアデバイスに接続するクライアントデバイス用の、クライアントデバイス認証ポリシー。各承認ポリシーは、クライアントデバイスがこれらのアクションを実行できるアクションセットとリソースを指定しています。

このオブジェクトには、次の情報が含まれます。

`policyNameKey`

この承認ポリシーの名前。を、この承認ポリシーを識別するのに役立つ名前*`policyNameKey`*に置き換えます。このポリシー名を使用して、デバイスグループに適用するポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

`statementNameKey`

このポリシーステートメントの名前。を、このポリシーステートメントを識別するのに役立つ名前*`statementNameKey`*に置き換えます。

このオブジェクトには、次の情報が含まれます。

`operations`

このポリシーのリソースに許可する操作のリスト。

以下のすべての操作を含めることができます。

- `mqtt:connect` - コアデバイスに接続するアクセス許可を付与します。コアデバイスに接続するには、クライアントデバイスにこのアクセス許可が必要です。

この操作は、次のリソースをサポートします。

- `mqtt:clientId:`*`deviceClientId`* - クライアントデバイスがコアデバイスの MQTT ブローカーへの接続に使用するクライアント ID に基づいて、アクセスを制限します。を使用するクライアント ID *`deviceClientId`*に置き換えます。

- `mqtt:publish` - MQTT メッセージをトピックにパブリッシュする権限を付与します。

  この操作は、次のリソースをサポートします。

  - `mqtt:topic:`*`mqttTopic`* - クライアントデバイスがメッセージをパブリッシュする MQTT トピックに基づいて、アクセスを制限します。*`mqttTopic`* を使用するトピックに置き換えます。

    このリソースは MQTT トピックのワイルドカードをサポートしていません。

- `mqtt:subscribe` - メッセージを受信する MQTT トピックフィルターにサブスクライブする許可を付与します。

  この操作は、次のリソースをサポートします。

  - `mqtt:topicfilter:`*`mqttTopicFilter`* - クライアントデバイスがメッセージにサブスクライブできる MQTT トピックに基づいて、アクセスを制限します。を使用するトピックフィルター*`mqttTopicFilter`*に置き換えます。

    このリソースは + および # MQTT トピックのワイルドカードをサポートしています。詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT トピック」を参照してください。

    クライアントデバイスは、許可を付与したトピックフィルターにのみサブスクライブできます。例えば、クライアントデバイスに `mqtt:topicfilter:client/+/status` リソースへのサブスクライブを許可した場合、クライアントデバイスは `client/+/status` にサブスクライブできますが、`client/client1/status` にはサブスクライブできません。

* ワイルドカードを指定すれば、すべてのアクションへのアクセスを許可することができます。

`resources`

このポリシーでの操作を許可するリソースのリスト。このポリシーでの操作に対応するリソースを指定します。例えば、`mqtt:publish` 操作を指定するポリシーで、MQTT トピックリソース (`mqtt:topic:`*`mqttTopic`*) のリストを指定することができます。

`*` ワイルドカードを指定すれば、すべてのリソースへのアクセスを許可することができます。`*` ワイルドカードを使用して、部分的なリソース識別子と一致させることはできません。例えば、**`"resources": "*"`** と指定することはできますが、**`"resources": "mqtt:clientId:*"`** と指定することはできません。

`statementDescription`

(オプション) このポリシーステートメントの説明。

`certificates`

(オプション) このコアデバイスの証明書設定オプション。このオブジェクトには、次の情報が含まれます。

`serverCertificateValiditySeconds`

(オプション) ローカル MQTT サーバー証明書の有効期限が切れるまでの時間 (秒単位)。このオプションを設定すれば、クライアントデバイスがコアデバイスから切断し、また再接続する回数をカスタマイズすることができます。

このコンポーネントは、有効期限が切れる 24 時間前に、ローカル MQTT サーバー証明書をローテーションします。Moquette MQTT ブローカーコンポーネントなどの MQTT ブローカーで、新しい証明書を生成して再起動します。これが実行されると、このコアデバイスに接続されているすべてのクライアントデバイスが切断されます。クライアントデバイスは、短時間待機した後に、コアデバイスに再度接続できます。

デフォルト: `604800` (7 日)

最小値: `172800` (2 日)

最大値: `864000` (10 日)

`performance`

(オプション) このコアデバイスのパフォーマンス設定オプション。このオブジェクトには、次の情報が含まれます。

`maxActiveAuthTokens`

(オプション) アクティブなクライアントデバイス認証トークンの最大数。この数を増やすと、より多くのクライアントデバイスが、再認証せずに単一のコアデバイスに接続できるようになります。

デフォルト: `2500`

`cloudRequestQueueSize`

(オプション) このコンポーネントで拒否される前に、キューに格納される AWS クラウドリクエストの最大数。

デフォルト: `100`

`maxConcurrentCloudRequests`

(オプション) AWS クラウド に同時に送信できるリクエストの最大数。この数を増やすと、多数のクライアントデバイスを接続するコアデバイスで、認証のパフォーマンスを向上させることができます。

デフォルト: `1`

`certificateAuthority`

(オプション) コアデバイスの中間認証機関を独自の中間認証機関に置き換える認証機関設定オプション。このオブジェクトには、次の情報が含まれます。

certificateUri

証明書の位置。ファイルシステム URI、またはハードウェアセキュリティモジュールに保存されている証明書をポイントする URI にすることができます。

`certificateChainUri`

コアデバイス CA の証明書チェーンの場所。これは、ルート CA に対する完全な証明書チェーンとなります。ファイルシステム URI、またはハードウェアセキュリティモジュールに保存されている証明書チェーンをポイントする URI にすることができます。

`privateKeyUri`

コアデバイスのプライベートキーの場所。これは、ファイルシステム URI、またはハードウェアセキュリティモジュールに保存されている証明書プライベートキーをポイントする URI にすることができます。

`security`

(オプション) このコアデバイスのセキュリティ設定オプション。このオブジェクトには、次の情報が含まれます。

`clientDeviceTrustDurationMinutes`

クライアントデバイスの認証情報を信頼できる時間 (分)。この時間が経過すると、コアデバイスでの再認証が必要となります。デフォルト値は 1 です。

Example 例: 設定マージの更新 (制限ポリシーを使用)

次の設定例では、名前が `MyClientDevice` で始まるクライアントデバイスに対して、すべてのトピックの接続とパブリッシュ/サブスクライブを許可するように指定しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
```

```
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example 例: 設定マージの更新 (許容ポリシーを使用)

次の設定例では、すべてのクライアントデバイスに対して、すべてのトピックに接続しパブリッシュ/サブスクライブすることを許可しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
      }
    },
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
```

```
  }
}
```

v2.2.x

`deviceGroups`

デバイスグループは、コアデバイスに接続して通信する権限を持つクライアントデバイスのグループです。選択ルールを使用してクライアントデバイスのグループを特定し、各デバイスグループにアクセス許可を指定する、クライアントデバイス認証ポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

`formatVersion`

この設定オブジェクトの形式バージョン。

次のオプションから選択します。

- `2021-03-05`

`definitions`

このコアデバイスのデバイスグループ。各定義では、クライアントデバイスがグループのメンバーであるかどうかを評価する選択ルールを指定します。各定義では、選択ルールに一致するクライアントデバイスに適用するアクセス権限ポリシーも指定します。クライアントデバイスが複数のデバイスグループのメンバーである場合、デバイスのアクセス許可は各グループのアクセス許可ポリシーで設定されます。

このオブジェクトには、次の情報が含まれます。

`groupNameKey`

このデバイスグループの名前。を、このデバイスグループを識別するのに役立つ名前*`groupNameKey`*に置き換えます。

このオブジェクトには、次の情報が含まれます。

`selectionRule`

このデバイスグループのメンバーであるクライアントデバイスを指定するクエリ。クライアントデバイスが接続されると、コアデバイスはこの選択ルールを評価して、クライアントデバイスがこのデバイスグループのメンバーであるかどうかを判断します。クライアントデバイスがメンバーである場合、コアデバイスはこのデバ

イスグループのポリシーを使用して、クライアントデバイスのアクションを承認します。

各選択ルールは少なくとも 1 つの選択ルール句で設定されています。この句は、クライアントデバイスと一致させることができる 1 つの式クエリです。選択ルールは、AWS IoT フリートのインデックス作成と同じクエリ構文を使用します。選択ルールの構文に関する詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT フリートのインデックス作成クエリ構文」を参照してください。

* ワイルドカードを使用して、複数のクライアントデバイスを 1 つの選択ルール句と一致させます。モノ名の末尾にこのワイルドカードを使用すると、指定した文字列で始まる名前のクライアントデバイスと一致させることができます。このワイルドカードを使用して、すべてのクライアントデバイスと一致させることもできます。

> **Note**
>
> コロン文字 (:) を含む値を選択する場合は、コロン文字にバックスラッシュ文字 (\\) を使用してエスケープします。JSON などの形式では、バックスラッシュ文字をエスケープする必要があるため、コロン文字の前に 2 つのバックスラッシュ文字を入力する必要があります。例えば、`thingName: MyTeam\\\\:ClientDevice1` が `MyTeam:ClientDevice1` という名前のモノを選択するように指定する場合です。

次のセレクターを指定できます。

- `thingName` - クライアントデバイスの AWS IoT モノの名前。

Example 選択ルール例

次の選択ルールは、名前が `MyClientDevice1` または `MyClientDevice2` のクライアントデバイスと一致します。

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example 選択ルール例 (ワイルドカードを使用)

次の選択ルールは、名前が `MyClientDevice` で始まるクライアントデバイスと一致します。

```
thingName: MyClientDevice*
```

Example 選択ルール例 (すべてのデバイスに一致)

次の選択ルールは、すべてのクライアントデバイスと一致します。

```
thingName: *
```

`policyName`

このデバイスグループ内のクライアントデバイスに適用されるアクセス許可ポリシー。`policies` オブジェクトで定義するポリシーの名前を指定します。

`policies`

コアデバイスに接続するクライアントデバイス用の、クライアントデバイス認証ポリシー。各承認ポリシーは、クライアントデバイスがこれらのアクションを実行できるアクションセットとリソースを指定しています。

このオブジェクトには、次の情報が含まれます。

`policyNameKey`

この承認ポリシーの名前。を、この承認ポリシーを識別するのに役立つ名前*`policyNameKey`*に置き換えます。このポリシー名を使用して、デバイスグループに適用するポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

`statementNameKey`

このポリシーステートメントの名前。を、このポリシーステートメントを識別するのに役立つ名前*`statementNameKey`*に置き換えます。

このオブジェクトには、次の情報が含まれます。

`operations`

このポリシーのリソースに許可する操作のリスト。

以下のすべての操作を含めることができます。

- `mqtt:connect` - コアデバイスに接続するアクセス許可を付与します。コアデバイスに接続するには、クライアントデバイスにこのアクセス許可が必要です。

この操作は、次のリソースをサポートします。

- `mqtt:clientId:`*`deviceClientId`* - クライアントデバイスがコアデバイスの MQTT ブローカーへの接続に使用するクライアント ID に基づいて、アクセスを制限します。を使用するクライアント ID *`deviceClientId`*に置き換えます。

- `mqtt:publish` - MQTT メッセージをトピックにパブリッシュする権限を付与します。

  この操作は、次のリソースをサポートします。

  - `mqtt:topic:`*`mqttTopic`* - クライアントデバイスがメッセージをパブリッシュする MQTT トピックに基づいて、アクセスを制限します。*`mqttTopic`* を使用するトピックに置き換えます。

    このリソースは MQTT トピックのワイルドカードをサポートしていません。

- `mqtt:subscribe` - メッセージを受信する MQTT トピックフィルターにサブスクライブする許可を付与します。

  この操作は、次のリソースをサポートします。

  - `mqtt:topicfilter:`*`mqttTopicFilter`* - クライアントデバイスがメッセージにサブスクライブできる MQTT トピックに基づいて、アクセスを制限します。を使用するトピックフィルター*`mqttTopicFilter`*に置き換えます。

    このリソースは + および # MQTT トピックのワイルドカードをサポートしています。詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT トピック」を参照してください。

    クライアントデバイスは、許可を付与したトピックフィルターにのみサブスクライブできます。例えば、クライアントデバイスに `mqtt:topicfilter:client/+/status` リソースへのサブスクライブを許可した場合、クライアントデバイスは `client/+/status` にサブスクライブできますが、`client/client1/status` にはサブスクライブできません。

* ワイルドカードを指定すれば、すべてのアクションへのアクセスを許可することができます。

`resources`

このポリシーでの操作を許可するリソースのリスト。このポリシーでの操作に対応するリソースを指定します。例えば、`mqtt:publish` 操作を指定するポリシーで、MQTT トピックリソース (`mqtt:topic:`*`mqttTopic`*) のリストを指定することができます。

`*` ワイルドカードを指定すれば、すべてのリソースへのアクセスを許可することができます。`*` ワイルドカードを使用して、部分的なリソース識別子と一致させることはできません。例えば、**`"resources": "*"`** と指定することはできますが、**`"resources": "mqtt:clientId:*"`** と指定することはできません。

`statementDescription`

(オプション) このポリシーステートメントの説明。

`certificates`

(オプション) このコアデバイスの証明書設定オプション。このオブジェクトには、次の情報が含まれます。

`serverCertificateValiditySeconds`

(オプション) ローカル MQTT サーバー証明書の有効期限が切れるまでの時間 (秒単位)。このオプションを設定すれば、クライアントデバイスがコアデバイスから切断し、また再接続する回数をカスタマイズすることができます。

このコンポーネントは、有効期限が切れる 24 時間前に、ローカル MQTT サーバー証明書をローテーションします。Moquette MQTT ブローカーコンポーネントなどの MQTT ブローカーで、新しい証明書を生成して再起動します。これが実行されると、このコアデバイスに接続されているすべてのクライアントデバイスが切断されます。クライアントデバイスは、短時間待機した後に、コアデバイスに再度接続できます。

デフォルト: `604800` (7 日)

最小値: `172800` (2 日)

最大値: `864000` (10 日)

`performance`

(オプション) このコアデバイスのパフォーマンス設定オプション。このオブジェクトには、次の情報が含まれます。

`maxActiveAuthTokens`

(オプション) アクティブなクライアントデバイス認証トークンの最大数。この数を増やすと、より多くのクライアントデバイスが、再認証せずに単一のコアデバイスに接続できるようになります。

デフォルト: `2500`

`cloudRequestQueueSize`

(オプション) このコンポーネントで拒否される前に、キューに格納される AWS クラウドリクエストの最大数。

デフォルト: `100`

`maxConcurrentCloudRequests`

(オプション) AWS クラウド に同時に送信できるリクエストの最大数。この数を増やすと、多数のクライアントデバイスを接続するコアデバイスで、認証のパフォーマンスを向上させることができます。

デフォルト: `1`

Example 例: 設定マージの更新 (制限ポリシーを使用)

次の設定例では、名前が `MyClientDevice` で始まるクライアントデバイスに対して、すべてのトピックの接続とパブリッシュ/サブスクライブを許可するように指定しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
```

```
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example 例: 設定マージの更新 (許容ポリシーを使用)

次の設定例では、すべてのクライアントデバイスに対して、すべてのトピックに接続しパブリッシュ/サブスクライブすることを許可しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
      }
    },
```

```
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

v2.1.x

`deviceGroups`

デバイスグループは、コアデバイスに接続して通信する権限を持つクライアントデバイスのグループです。選択ルールを使用してクライアントデバイスのグループを特定し、各デバイスグループにアクセス許可を指定する、クライアントデバイス認証ポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

`formatVersion`

この設定オブジェクトの形式バージョン。

次のオプションから選択します。

- `2021-03-05`

`definitions`

このコアデバイスのデバイスグループ。各定義では、クライアントデバイスがグループのメンバーであるかどうかを評価する選択ルールを指定します。各定義では、選択ルールに一致するクライアントデバイスに適用するアクセス権限ポリシーも指定します。クライアントデバイスが複数のデバイスグループのメンバーである場合、デバイスのアクセス許可は各グループのアクセス許可ポリシーで設定されます。

このオブジェクトには、次の情報が含まれます。

groupNameKey

このデバイスグループの名前。を、このデバイスグループを識別するのに役立つ名前*groupNameKey*に置き換えます。

このオブジェクトには、次の情報が含まれます。

`selectionRule`

このデバイスグループのメンバーであるクライアントデバイスを指定するクエリ。クライアントデバイスが接続されると、コアデバイスはこの選択ルールを評価して、クライアントデバイスがこのデバイスグループのメンバーであるかどうかを判断します。クライアントデバイスがメンバーである場合、コアデバイスはこのデバイスグループのポリシーを使用して、クライアントデバイスのアクションを承認します。

各選択ルールは少なくとも 1 つの選択ルール句で設定されています。この句は、クライアントデバイスと一致させることができる 1 つの式クエリです。選択ルールは、AWS IoT フリートのインデックス作成と同じクエリ構文を使用します。選択ルールの構文に関する詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT フリートのインデックス作成クエリ構文」を参照してください。

* ワイルドカードを使用して、複数のクライアントデバイスを 1 つの選択ルール句と一致させます。モノ名の末尾にこのワイルドカードを使用すると、指定した文字列で始まる名前のクライアントデバイスと一致させることができます。このワイルドカードを使用して、すべてのクライアントデバイスと一致させることもできます。

> **Note**
>
> コロン文字 (`:`) を含む値を選択する場合は、コロン文字にバックスラッシュ文字 (`\\`) を使用してエスケープします。JSON などの形式では、バックスラッシュ文字をエスケープする必要があるため、コロン文字の前に 2 つのバックスラッシュ文字を入力する必要があります。例えば、`thingName:MyTeam\\\\:ClientDevice1` が `MyTeam:ClientDevice1` という名前のモノを選択するように指定する場合です。

次のセレクターを指定できます。

- `thingName` - クライアントデバイスの AWS IoT モノの名前。

Example 選択ルール例

次の選択ルールは、名前が `MyClientDevice1` または `MyClientDevice2` のクライアントデバイスと一致します。

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example 選択ルール例 (ワイルドカードを使用)

次の選択ルールは、名前が `MyClientDevice` で始まるクライアントデバイスと一致します。

```
thingName: MyClientDevice*
```

Example 選択ルール例 (すべてのデバイスに一致)

次の選択ルールは、すべてのクライアントデバイスと一致します。

```
thingName: *
```

`policyName`

このデバイスグループ内のクライアントデバイスに適用されるアクセス許可ポリシー。`policies` オブジェクトで定義するポリシーの名前を指定します。

`policies`

コアデバイスに接続するクライアントデバイス用の、クライアントデバイス認証ポリシー。各承認ポリシーは、クライアントデバイスがこれらのアクションを実行できるアクションセットとリソースを指定しています。

このオブジェクトには、次の情報が含まれます。

`policyNameKey`

この承認ポリシーの名前。を、この承認ポリシーを識別するのに役立つ名前*`policyNameKey`*に置き換えます。このポリシー名を使用して、デバイスグループに適用するポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

statementNameKey

このポリシーステートメントの名前。を、このポリシーステートメントを識別するのに役立つ名前*statementNameKey*に置き換えます。

このオブジェクトには、次の情報が含まれます。

operations

このポリシーのリソースに許可する操作のリスト。

以下のすべての操作を含めることができます。

- mqtt:connect - コアデバイスに接続するアクセス許可を付与します。コアデバイスに接続するには、クライアントデバイスにこのアクセス許可が必要です。

  この操作は、次のリソースをサポートします。

  - mqtt:clientId:*deviceClientId* - クライアントデバイスがコアデバイスの MQTT ブローカーへの接続に使用するクライアント ID に基づいて、アクセスを制限します。を使用するクライアント ID *deviceClientId*に置き換えます。
- mqtt:publish - MQTT メッセージをトピックにパブリッシュする権限を付与します。

  この操作は、次のリソースをサポートします。

  - mqtt:topic:*mqttTopic* - クライアントデバイスがメッセージをパブリッシュする MQTT トピックに基づいて、アクセスを制限します。*mqttTopic* を使用するトピックに置き換えます。

    このリソースは MQTT トピックのワイルドカードをサポートしていません。
- mqtt:subscribe - メッセージを受信する MQTT トピックフィルターにサブスクライブする許可を付与します。

  この操作は、次のリソースをサポートします。

  - mqtt:topicfilter:*mqttTopicFilter* - クライアントデバイスがメッセージにサブスクライブできる MQTT トピックに基づいて、アクセスを制

限します。を使用するトピックフィルター*mqttTopicFilter*に置き換えます。

このリソースは + および # MQTT トピックのワイルドカードをサポートしています。詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT トピック」を参照してください。

クライアントデバイスは、許可を付与したトピックフィルターにのみサブスクライブできます。例えば、クライアントデバイスに `mqtt:topicfilter:client/+/status` リソースへのサブスクライブを許可した場合、クライアントデバイスは `client/+/status` にサブスクライブできますが、`client/client1/status` にはサブスクライブできません。

* ワイルドカードを指定すれば、すべてのアクションへのアクセスを許可することができます。

`resources`

このポリシーでの操作を許可するリソースのリスト。このポリシーでの操作に対応するリソースを指定します。例えば、`mqtt:publish` 操作を指定するポリシーで、MQTT トピックリソース (`mqtt:topic:`*mqttTopic*) のリストを指定することができます。

* ワイルドカードを指定すれば、すべてのリソースへのアクセスを許可することができます。* ワイルドカードを使用して、部分的なリソース識別子と一致させることはできません。例えば、**`"resources": "*"`** と指定することはできますが、**`"resources": "mqtt:clientId:*"`** と指定することはできません。

`statementDescription`

(オプション) このポリシーステートメントの説明。

`certificates`

(オプション) このコアデバイスの証明書設定オプション。このオブジェクトには、次の情報が含まれます。

`serverCertificateValiditySeconds`

(オプション) ローカル MQTT サーバー証明書の有効期限が切れるまでの時間 (秒単位)。このオプションを設定すれば、クライアントデバイスがコアデバイスから切断し、また再接続する回数をカスタマイズすることができます。

このコンポーネントは、有効期限が切れる 24 時間前に、ローカル MQTT サーバー証明書をローテーションします。Moquette MQTT ブローカーコンポーネントなどの MQTT ブローカーで、新しい証明書を生成して再起動します。これが実行されると、このコアデバイスに接続されているすべてのクライアントデバイスが切断されます。クライアントデバイスは、短時間待機した後に、コアデバイスに再度接続できます。

デフォルト: `604800` (7 日)

最小値: `172800` (2 日)

最大値: `864000` (10 日)

Example 例: 設定マージの更新 (制限ポリシーを使用)

次の設定例では、名前が `MyClientDevice` で始まるクライアントデバイスに対して、すべてのトピックの接続とパブリッシュ/サブスクライブを許可するように指定しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
```

```
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example 例: 設定マージの更新 (許容ポリシーを使用)

次の設定例では、すべてのクライアントデバイスに対して、すべてのトピックに接続しパブリッシュ/サブスクライブすることを許可しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
      }
    },
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
          "resources": [
            "*"
          ]
```

```
        }
      }
    }
  }
}
```

v2.0.x

`deviceGroups`

デバイスグループは、コアデバイスに接続して通信する権限を持つクライアントデバイスのグループです。選択ルールを使用してクライアントデバイスのグループを特定し、各デバイスグループにアクセス許可を指定する、クライアントデバイス認証ポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

`formatVersion`

この設定オブジェクトの形式バージョン。

次のオプションから選択します。

- `2021-03-05`

`definitions`

このコアデバイスのデバイスグループ。各定義では、クライアントデバイスがグループのメンバーであるかどうかを評価する選択ルールを指定します。各定義では、選択ルールに一致するクライアントデバイスに適用するアクセス権限ポリシーも指定します。クライアントデバイスが複数のデバイスグループのメンバーである場合、デバイスのアクセス許可は各グループのアクセス許可ポリシーで設定されます。

このオブジェクトには、次の情報が含まれます。

`groupNameKey`

このデバイスグループの名前。を、このデバイスグループを識別するのに役立つ名前*`groupNameKey`*に置き換えます。

このオブジェクトには、次の情報が含まれます。

`selectionRule`

このデバイスグループのメンバーであるクライアントデバイスを指定するクエリ。クライアントデバイスが接続されると、コアデバイスはこの選択ルールを評価し

て、クライアントデバイスがこのデバイスグループのメンバーであるかどうかを判断します。クライアントデバイスがメンバーである場合、コアデバイスはこのデバイスグループのポリシーを使用して、クライアントデバイスのアクションを承認します。

各選択ルールは少なくとも 1 つの選択ルール句で設定されています。この句は、クライアントデバイスと一致させることができる 1 つの式クエリです。選択ルールは、AWS IoT フリートのインデックス作成と同じクエリ構文を使用します。選択ルールの構文に関する詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT フリートのインデックス作成クエリ構文」を参照してください。

* ワイルドカードを使用して、複数のクライアントデバイスを 1 つの選択ルール句と一致させます。モノ名の末尾にこのワイルドカードを使用すると、指定した文字列で始まる名前のクライアントデバイスと一致させることができます。このワイルドカードを使用して、すべてのクライアントデバイスと一致させることもできます。

> **Note**
>
> コロン文字 (:) を含む値を選択する場合は、コロン文字にバックスラッシュ文字 (\\) を使用してエスケープします。JSON などの形式では、バックスラッシュ文字をエスケープする必要があるため、コロン文字の前に 2 つのバックスラッシュ文字を入力する必要があります。例えば、`thingName: MyTeam\\\\:ClientDevice1` が `MyTeam:ClientDevice1` という名前のモノを選択するように指定する場合です。

次のセレクターを指定できます。

- `thingName` - クライアントデバイスの AWS IoT モノの名前。

Example 選択ルール例

次の選択ルールは、名前が `MyClientDevice1` または `MyClientDevice2` のクライアントデバイスと一致します。

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example 選択ルール例 (ワイルドカードを使用)

次の選択ルールは、名前が MyClientDevice で始まるクライアントデバイスと一致します。

```
thingName: MyClientDevice*
```

Example 選択ルール例 (すべてのデバイスに一致)

次の選択ルールは、すべてのクライアントデバイスと一致します。

```
thingName: *
```

`policyName`

このデバイスグループ内のクライアントデバイスに適用されるアクセス許可ポリシー。policies オブジェクトで定義するポリシーの名前を指定します。

`policies`

コアデバイスに接続するクライアントデバイス用の、クライアントデバイス認証ポリシー。各承認ポリシーは、クライアントデバイスがこれらのアクションを実行できるアクションセットとリソースを指定しています。

このオブジェクトには、次の情報が含まれます。

`policyNameKey`

この承認ポリシーの名前。を、この承認ポリシーを識別するのに役立つ名前*`policyNameKey`*に置き換えます。このポリシー名を使用して、デバイスグループに適用するポリシーを定義します。

このオブジェクトには、次の情報が含まれます。

`statementNameKey`

このポリシーステートメントの名前。を、このポリシーステートメントを識別するのに役立つ名前*`statementNameKey`*に置き換えます。

このオブジェクトには、次の情報が含まれます。

`operations`

このポリシーのリソースに許可する操作のリスト。

以下のすべての操作を含めることができます。

- `mqtt:connect` - コアデバイスに接続するアクセス許可を付与します。コアデバイスに接続するには、クライアントデバイスにこのアクセス許可が必要です。

  この操作は、次のリソースをサポートします。

  - `mqtt:clientId:`*`deviceClientId`* - クライアントデバイスがコアデバイスの MQTT ブローカーへの接続に使用するクライアント ID に基づいて、アクセスを制限します。を使用するクライアント ID *`deviceClientId`*に置き換えます。

- `mqtt:publish` - MQTT メッセージをトピックにパブリッシュする権限を付与します。

  この操作は、次のリソースをサポートします。

  - `mqtt:topic:`*`mqttTopic`* - クライアントデバイスがメッセージをパブリッシュする MQTT トピックに基づいて、アクセスを制限します。*`mqttTopic`* を使用するトピックに置き換えます。

    このリソースは MQTT トピックのワイルドカードをサポートしていません。

- `mqtt:subscribe` - メッセージを受信する MQTT トピックフィルターにサブスクライブする許可を付与します。

  この操作は、次のリソースをサポートします。

  - `mqtt:topicfilter:`*`mqttTopicFilter`* - クライアントデバイスがメッセージにサブスクライブできる MQTT トピックに基づいて、アクセスを制限します。を使用するトピックフィルター*`mqttTopicFilter`*に置き換えます。

    このリソースは + および # MQTT トピックのワイルドカードをサポートしています。詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT トピック」を参照してください。

    クライアントデバイスは、許可を付与したトピックフィルターにのみサブスクライブできます。例えば、クライアントデバイスに `mqtt:topicfilter:client/+/status` リソースへのサブスクライブを許可した場合、クライアントデバイスは `client/+/status` にサブスクラ

イブできますが、client/client1/status にはサブスクライブできません。

* ワイルドカードを指定すれば、すべてのアクションへのアクセスを許可することができます。

`resources`

このポリシーでの操作を許可するリソースのリスト。このポリシーでの操作に対応するリソースを指定します。例えば、mqtt:publish 操作を指定するポリシーで、MQTT トピックリソース (mqtt:topic:*mqttTopic*) のリストを指定することができます。

* ワイルドカードを指定すれば、すべてのリソースへのアクセスを許可することができます。* ワイルドカードを使用して、部分的なリソース識別子と一致させることはできません。例えば、**"resources": "*"** と指定することはできますが、**"resources": "mqtt:clientId:*"** と指定することはできません。

`statementDescription`

(オプション) このポリシーステートメントの説明。

Example 例: 設定マージの更新 (制限ポリシーを使用)

次の設定例では、名前が MyClientDevice で始まるクライアントデバイスに対して、すべてのトピックの接続とパブリッシュ/サブスクライブを許可するように指定しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
```

```
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example 例: 設定マージの更新 (許容ポリシーを使用)

次の設定例では、すべてのクライアントデバイスに対して、すべてのトピックに接続しパブリッシュ/サブスクライブすることを許可しています。

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
      }
    },
    "policies": {
```

```
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.4.5	新機能 `selectionRule` パラメータでモノの名前を選択するためのワイルドカードプレフィックスのサポートを追加しました。 バグ修正と機能向上 特定のケースで証明書が新しい接続情報で更新されない問題を修正しました。
2.4.4	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.4.3	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.4.2	新機能 新しい `startupTimeoutSeconds` 構成オプションを追加します。
2.4.1	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.4.0	新機能 • テレメトリエージェントによって公開される操作メトリックを出力するためのクライアントデバイス認証のサポートを追加します。

バージョン	変更
	バグ修正と機能向上 • クライアントデバイスの認証がクライアントデバイスの ID を確認するのに 10 秒以上かかる問題を修正しました。 • 追加のマイナー修正と機能向上。
2.3.2	バグ修正と機能向上 • 新たに、ホスト名情報のキャッシュ機能がサポートされ、オフラインでの再起動時に、コンポーネントが正しい証明書サブジェクトを生成するようになりました。
2.3.1	バグ修正と機能向上 • メモリリークを修正しました。
2.3.0	**Warning** このバージョンは現在利用できません。このバージョンの改善は、このコンポーネントのそれ以降のバージョンで利用できます。 新機能 • コアデバイスがインターネットに接続されていない場合でも引き続きコアデバイスに接続できるようにするために、クライアントデバイスのオフライン認証のサポートを追加しました。 • コアデバイスが MQTT ブローカー証明書を生成するためのルート証明書として使用する、お客様が提供する認証機関のサポートを追加しました。
2.2.3	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.2.2	バグ修正と機能向上 • 特定のシナリオで、ローカル MQTT サーバー証明書が意図した回数よりも頻繁にローテーションされる問題を修正しました。

バージョン	変更
2.2.1	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.2.0	新機能 • プロセス間通信 (IPC) オペレーションを呼び出して、クライアントデバイスを認証および承認する、カスタムコンポーネントのサポートを追加しました。例えば、カスタム MQTT ブローカーコンポーネントで、これらのオペレーションを使用できます。詳細については、「IPC: Authenticate and authorize client devices」(IPC: クライアントデバイスの認証と承認) を参照してください。 • このコンポーネントの処理内容を調整するための、`maxActiveAuthTokens`、`cloudQueueSize`、および `threadPoolSize` オプションを追加しました。
2.1.0	新機能 • MQTT ブローカーサーバー証明書の有効期限が切れたときに、カスタマイズするために設定できる `serverCertificateValiditySeconds` オプションを追加しました。サーバー証明書は 2 ~ 10 日後に期限切れになるように設定できます。 バグ修正と機能向上 • このコンポーネントが設定リセット更新を処理する際に関する問題を修正しました。 • 特定のシナリオで、ローカル MQTT サーバー証明書が意図した回数よりも頻繁にローテーションされる問題を修正しました。 この修正を適用するには、Moquette MQTT ブローカーコンポーネントの v2.1.0 以降も使用する必要があります。 • 証明書をローテーションするときに、このコンポーネントがログに記録するメッセージを改善しました。 • Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.0.4	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.0.3	バグ修正と機能向上 • コアデバイスの秘密キーをローテーションすると、認証情報が更新されるようになりました。 • ログメッセージをより明確にするための更新。
2.0.2	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.0.1	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.0.0	当初のバージョン

CloudWatch メトリクス

Amazon CloudWatch メトリクスコンポーネント (`aws.greengrass.Cloudwatch`) は、Greengrass コアデバイスから Amazon にカスタムメトリクスを発行します CloudWatch。コンポーネントを使用すると、コンポーネントは CloudWatch メトリクスをパブリッシュできます。メトリクスは、Greengrass コアデバイスの環境のモニタリングと分析に使用できます。詳細については、「Amazon CloudWatch ユーザーガイド」の「Amazon メトリクスの使用 CloudWatch 」を参照してください。

このコンポーネントで CloudWatch メトリクスを発行するには、このコンポーネントがサブスクライブするトピックにメッセージを発行します。デフォルトでは、このコンポーネントは `cloudwatch/metric/put` ローカルパブリッシュ/サブスクライブトピックにサブスクライブします。このコンポーネントをデプロイするときに、 AWS IoT Core MQTT トピックを含む他のトピックを指定できます。

このコンポーネントは、同じ名前空間にあるメトリクスをバッチ処理し、 CloudWatch 一定の間隔でに発行します。

> **Note**
>
> このコンポーネントは、 AWS IoT Greengrass V1 の CloudWatch メトリクスコネクタと同様の機能を提供します。詳細については、AWS IoT Greengrass 「 V1 デベロッパーガイド」の「 CloudWatch メトリクスコネクタ」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 3.1.x
- 3.0.x
- 2.1.x
- 2.0.x

コンポーネントの各バージョンに対する変更については、「変更ログ」を参照してください。

タイプ

v3.x

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

v2.x

このコンポーネントは Lambda コンポーネントです (`aws.greengrass.lambda`)。Greengrass nucleus は、Lambda ランチャーコンポーネントを使用してこのコンポーネントの Lambda 関数を実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

v3.x

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

v2.x

このコンポーネントは、Linux コアデバイスにのみインストールできます。

要件

このコンポーネントには次の要件があります。

3.x

- Python バージョン 3.7 がコアデバイスにインストールされ、PATH 環境変数に追加されています。
- 次の IAM ポリシーの例で示されているように、Greengrass デバイスのロールは `cloudwatch:PutMetricData` アクションを許可する必要があります。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "cloudwatch:PutMetricData"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

詳細については、「Amazon CloudWatch ユーザーガイド」の「Amazon アクセス許可リファレンス CloudWatch 」を参照してください。

2.x

- コアデバイスは、Lambda 関数を実行するための要件を満たしている必要があります。コアデバイスが、コンテナ化された Lambda 関数を実行させる場合、そのデバイスは要件を満たす必要があります。詳細については、「Lambda 関数の要件」を参照してください。
- Python バージョン 3.7 がコアデバイスにインストールされ、PATH 環境変数に追加されています。
- 次の IAM ポリシーの例で示されているように、Greengrass デバイスのロールは `cloudwatch:PutMetricData` アクションを許可する必要があります。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "cloudwatch:PutMetricData"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

詳細については、「Amazon CloudWatch ユーザーガイド」の「Amazon アクセス許可リファレンス CloudWatch 」を参照してください。

- このコンポーネントから出力データを受信するには、このコンポーネントをデプロイするときに、次の設定更新プログラムをレガシーサブスクリプションルーターのコンポーネント (aws.greengrass.LegacySubscriptionRouter) のためにマージする必要があります。この設定は、このコンポーネントがレスポンスを公開するトピックを指定します。

  Legacy subscription router v2.1.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-cloudwatch": {
        "id": "aws-greengrass-cloudwatch",
        "source": "component:aws.greengrass.Cloudwatch",
        "subject": "cloudwatch/metric/put/status",
        "target": "cloud"
      }
    }
  }
  ```

  Legacy subscription router v2.0.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-cloudwatch": {
        "id": "aws-greengrass-cloudwatch",
        "source": "arn:aws:lambda:region:aws:function:aws-greengrass-
  cloudwatch:version",
        "subject": "cloudwatch/metric/put/status",
        "target": "cloud"
      }
    }
  }
  ```

  - *region* AWS リージョン は、使用する に置き換えます。
  - *#####*を、このコンポーネントが実行する Lambda 関数のバージョンに置き換えます。Lambda 関数のバージョンを確認するには、デプロイするこのコンポーネントのバージョンの recipe を確認する必要があります。AWS IoT Greengrass コンソールで、このコンポーネントの詳細ページを開き、[Lambda function] (Lambda 関数) の key-value ペアを見つけます。このキー値のペアには、Lambda 関数の名前とバージョンが含まれます。

> **Important**
>
> このコンポーネントをデプロイするたびに、レガシーサブスクリプションルーターの Lambda 関数のバージョンを更新する必要があります。これにより、デプロイするコンポーネントバージョンに正しい Lambda 関数のバージョンが使用されることが保証されます。

詳細については、「デプロイの作成」を参照してください。

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
`monitoring. region.amazonaws.com`	443	はい	CloudWatch メトリクスをアップロードします。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

3.0.0 - 3.1.0

次の表に、このコンポーネントのバージョン 3.0.0 から 3.1.0 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <3.0.0	ソフト
トークン交換サービス	>=0.0.0	ハード

2.1.2 and 2.1.3

次の表に、このコンポーネントのバージョン 2.1.2 および 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.8 - 2.1.0

次の表に、このコンポーネントのバージョン 2.0.8 から 2.1.0 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.7

次の表に、このコンポーネントのバージョン 2.0.7 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.6

次の表に、このコンポーネントのバージョン 2.0.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.5

次の表に、このコンポーネントのバージョン 2.0.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.4

次の表に、このコンポーネントのバージョン 2.0.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.3

次の表に、このコンポーネントのバージョン 2.0.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.3 <2.1.0	ハード
Lambda ランチャー	>=1.0.0	ハード
Lambda ランタイム	>=1.0.0	ソフト

依存関係	互換性のあるバージョン	依存関係タイプ
トークン交換サービス	>=1.0.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

v3.x

`PublishInterval`

(オプション) 特定の名前空間にメトリクスのバッチを発行するまでに待機する最大秒数。メトリクスを受信した時点でメトリクスをパブリッシュするようにコンポーネントを設定するには (バッチ処理なし)、`0` を指定します。

コンポーネントは、同じ名前空間で 20 個のメトリクスを受信 CloudWatch した後、または指定した間隔後に にパブリッシュします。

> **Note**
> コンポーネントは、イベントがパブリッシュされる順序を指定しません。

この値の最大値は 900 秒です。

デフォルト: 10 秒

`MaxMetricsToRetain`

(オプション) コンポーネントが新しいメトリクスに置き換わる前にメモリに保存される、すべての名前空間全体でのメトリクスの最大数です。

この制限はコアデバイスがインターネットに接続していないときに適用され、コンポーネントは後でパブリッシュするためにメトリクスをバッファします。バッファが満杯になると、コン

ポーネントは最も古いメトリクスを新しいメトリクスに置き換えます。特定の名前空間内のメトリクスは、同じ名前空間内のメトリクスにのみ置き換えられます。

> **Note**
>
> ホストによるコンポーネント処理が中断された場合、コンポーネントはメトリクスを保存しません。この状況は、例えばデプロイ中やコアデバイスの再起動時に発生する可能性があります。

この値は 2,000 メトリクス以上に指定する必要があります。

デフォルト: 5,000 メトリクス

`InputTopic`

(オプション) コンポーネントがメッセージを受信するためにサブスクライブするトピック。`PubSubToIoTCore` の `true` を指定した場合、このトピックで MQTT ワイルドカード (+ および #) を使用できます。

デフォルト: `cloudwatch/metric/put`

`OutputTopic`

(オプション) コンポーネントがステータスレスポンスをパブリッシュする先となるトピック。

デフォルト: `cloudwatch/metric/put/status`

`PubSubToIoTCore`

(オプション) AWS IoT Core MQTT トピックにパブリッシュしてサブスクライブするかどうかを定義する文字列値。サポートされている値は、`true` および `false` です。

デフォルト: `false`

`UseInstaller`

(オプション) このコンポーネントの SDK 依存関係をインストールするために、このコンポーネントでインストーラスクリプトを使用するかどうかを定義するブール値。

依存関係のインストールにカスタムスクリプトを使用する場合、またはビルド済みの Linux イメージにランタイムの依存関係を含める場合は、この値を `false` に設定します。このコン

ポーネントを使用するには、依存関係を含む次のライブラリをインストールし、デフォルトの Greengrass システムユーザーを利用できるようにする必要があります。

- AWS IoT Device SDK v2 for Python
- AWS SDK for Python (Boto3)

デフォルト: `true`

`PublishRegion`

(オプション) CloudWatch メトリクスを発行する AWS リージョン 先の 。この値は、コアデバイスのデフォルトのリージョンを上書きします。このパラメータは、クロスリージョンメトリクスにのみ必要です。

`accessControl`

(オプション) コンポーネントが指定されているトピックにパブリッシュしサブスクライブできるようにする承認ポリシーが含まれるオブジェクト。`InputTopic` および `OutputTopic` のカスタム値を指定した場合には、このオブジェクトのリソース値も更新する必要があります。

デフォルト:

```
{
  "aws.greengrass.ipc.pubsub": {
    "aws.greengrass.Cloudwatch:pubsub:1": {
      "policyDescription": "Allows access to subscribe to input topics.",
      "operations": [
        "aws.greengrass#SubscribeToTopic"
      ],
      "resources": [
        "cloudwatch/metric/put"
      ]
    },
    "aws.greengrass.Cloudwatch:pubsub:2": {
      "policyDescription": "Allows access to publish to output topics.",
      "operations": [
        "aws.greengrass#PublishToTopic"
      ],
      "resources": [
        "cloudwatch/metric/put/status"
      ]
    }
  },
  "aws.greengrass.ipc.mqttproxy": {
```

```
    "aws.greengrass.Cloudwatch:mqttproxy:1": {
      "policyDescription": "Allows access to subscribe to input topics.",
      "operations": [
        "aws.greengrass#SubscribeToIoTCore"
      ],
      "resources": [
        "cloudwatch/metric/put"
      ]
    },
    "aws.greengrass.Cloudwatch:mqttproxy:2": {
      "policyDescription": "Allows access to publish to output topics.",
      "operations": [
        "aws.greengrass#PublishToIoTCore"
      ],
      "resources": [
        "cloudwatch/metric/put/status"
      ]
    }
  }
}
```

Example 例: 設定マージの更新

```
{
  "PublishInterval": 0,
  "PubSubToIoTCore": true
}
```

v2.x

> **Note**
>
> このコンポーネントのデフォルト設定には、Lambda 関数のパラメータが含まれます。デバイスにこのコンポーネントを設定するには、次のパラメータのみを編集することをお勧めします。

`lambdaParams`

このコンポーネントの Lambda 関数のパラメータを含むオブジェクト。このオブジェクトには、次の情報が含まれます。

`EnvironmentVariables`

Lambda 関数のパラメータを含むオブジェクト。このオブジェクトには、次の情報が含まれます。

`PUBLISH_INTERVAL`

(オプション) 特定の名前空間にメトリクスのバッチを発行するまでに待機する最大秒数。メトリクスを受信した時点でメトリクスをパブリッシュするようにコンポーネントを設定するには (バッチ処理なし)、`0` を指定します。

コンポーネントは、同じ名前空間で 20 個のメトリクスを受信 CloudWatch した後、または指定した間隔後に にパブリッシュします。

> **Note**
>
> このコンポーネントは、イベントがパブリッシュされる順序を保証しません。

この値は最大 900 秒になります。

デフォルト: 10 秒

`MAX_METRICS_TO_RETAIN`

(オプション) コンポーネントが新しいメトリクスに置き換わる前にメモリに保存される、すべての名前空間全体でのメトリクスの最大数です。

この制限はコアデバイスがインターネットに接続していないときに適用され、コンポーネントは後でパブリッシュするためにメトリクスをバッファします。バッファが満杯になると、コンポーネントは最も古いメトリクスを新しいメトリクスに置き換えます。特定の名前空間内のメトリクスは、同じ名前空間内のメトリクスにのみ置き換えられます。

> **Note**
>
> ホストによるコンポーネント処理が中断された場合、コンポーネントはメトリクスを保存しません。この状況は、例えばデプロイ中やコアデバイスの再起動時に発生する可能性があります。

この値は 2,000 メトリクス以上に指定する必要があります。

デフォルト: 5,000 メトリクス

`PUBLISH_REGION`

(オプション) CloudWatch メトリクスを発行する AWS リージョン 先の 。この値は、コアデバイスのデフォルトのリージョンを上書きします。このパラメータは、クロスリージョンメトリクスにのみ必要です。

`containerMode`

(オプション) このコンポーネントのコンテナ化モード。次のオプションから選択します。

- `NoContainer` - コンポーネントは、分離されたランタイム環境では実行されません。
- `GreengrassContainer` – コンポーネントは、 AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。

デフォルト: `GreengrassContainer`

`containerParams`

(オプション) このコンポーネントのコンテナパラメータを含むオブジェクト。`containerMode` の `GreengrassContainer` を指定した場合、コンポーネントはこれらのパラメータを使用します。

このオブジェクトには、次の情報が含まれます。

`memorySize`

(オプション) コンポーネントに割り当てるメモリ量 (KB 単位)。

デフォルトは 64 MB (65,535 KB) です。

`pubsubTopics`

(オプション) コンポーネントがメッセージを受信するためにサブスクライブするトピックを含むオブジェクト。各トピックと、コンポーネントが から MQTT トピックをサブスクライブするか、ローカルのパブリッシュ/サブスクライブトピックをサブスクライブ AWS IoT Core するかを指定できます。

このオブジェクトには、次の情報が含まれます。

`0` - これは文字列としての配列インデックスです。

次の情報が含まれるオブジェクト。

`type`

(オプション) このコンポーネントがメッセージをサブスクライブするために使用するパブリッシュ/サブスクライブメッセージングのタイプ。次のオプションから選択します。

- PUB_SUB - ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブします。このオプションを選択した場合、トピックに MQTT ワイルドカードを含めることはできません。このオプションを指定したときに、カスタムコンポーネントからメッセージを送信する方法の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。
- IOT_CORE – AWS IoT Core MQTT メッセージをサブスクライブします。このオプションを選択した場合、トピックに MQTT ワイルドカードを含めることができます。このオプションを指定したときに、カスタムコンポーネントからメッセージを送信する方法の詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。

デフォルト: PUB_SUB

`topic`

(オプション) コンポーネントがメッセージを受信するためにサブスクライブするトピック。`type` の `IotCore` を指定した場合、このトピックで MQTT ワイルドカード (+ および #) を使用できます。

Example 例: 設定マージの更新 (コンテナモード)

```
{
  "containerMode": "GreengrassContainer"
}
```

Example 例: 設定マージの更新 (コンテナモードなし)

```
{
  "containerMode": "NoContainer"
}
```

入力データ

このコンポーネントは、次のトピックのメトリクスを受け入れ、メトリクスを に発行します CloudWatch。デフォルトで、このコンポーネントはローカルのパブリッシュ/サブスクライブメッセージにサブスクライブします。カスタムコンポーネントからこのコンポーネントにメッセージをパブリッシュする方法の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。

コンポーネントバージョン v3.0.0 以降では、オプションで PubSubToIoTCore 設定パラメータを true に設定することで、MQTT トピックをサブスクライブするようにこのコンポーネントを設定することができます。カスタムコンポーネントで MQTT トピックにメッセージをパブリッシュする方法の詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。

既定のトピック: `cloudwatch/metric/put`

メッセージは、次のプロパティを受付けます。入力メッセージは JSON 形式である必要があります。

`request`

このメッセージのメトリクス。

リクエストオブジェクトには、 CloudWatch に発行するメトリクスデータが含まれています。メトリクス値は、 `PutMetricData` 動作の仕様を満たしている必要があります。

タイプ: 次の情報が含まれる `object`。

`namespace`

この request. CloudWatch uses のメトリクスデータのユーザー定義の名前空間は、メトリクスデータポイントのコンテナとして使用されます。

> **Note**
>
> 予約済みの文字列 `AWS/` で始まる名前空間を指定することはできません。

タイプ: `string`

有効なパターン: `[^:].*`

`metricData`

メトリクスのデータ。

タイプ: 次の情報が含まれる `object`。

`metricName`

メトリクスの名前。

タイプ: `string`

`value`

メトリクスの値。

> **Note**
>
> CloudWatch は、小さすぎる、または大きすぎる値を拒否します。値は、`8.515920e-109`と `1.174271e+108` (ベース 10) または `2e-360`と `2e360` (ベース 2) の間である必要があります。 CloudWatch は、`NaN`、、 `+Infinity`などの特殊な値をサポートしていません`-Infinity`。

タイプ: `double`

`dimensions`

(オプション) メトリクスのディメンション。ディメンションは、メトリクスとそのデータに関する追加情報を提供します。1 つのメトリクスでは、最大 10 個のディメンションを定義できます。

このコネクタには `coreName` という名前のディメンションが自動的に含まれ、そのコアデバイスの名前を値に持ちます。

タイプ: `array` のオブジェクトであり、各々に次の情報が含まれます。

`name`

(オプション) ディメンション名。

タイプ: `string`

`value`

(オプション) ディメンション値。

タイプ: `string`

`timestamp`

(オプション) データが受信された時間。UNIX エポック時間で秒単位で表されます。

デフォルトは、コンポーネントがメッセージを受信した時間です。

タイプ: `double`

> **Note**
>
> このコンポーネントのバージョン 2.0.3 ~ 2.0.7 を使用する場合は、1 つのソースから複数のメトリクスを送信するときに、メトリクスごとに個別にタイムスタンプを取得することをお勧めします。タイムスタンプを保存するのに変数を使用してしないでください。

`unit`

(オプション) メトリクスの単位。

タイプ: `string`

有効な値:
`Seconds`、`Microseconds`、`Milliseconds`、`Bytes`、`Kilobytes`、`Megabytes`、`Gigabytes`、`Terabytes`、`Bits`、`Kilobits`、`Megabits`、`Gigabits`、`Terabits`、`Count`、`None`、`Bytes/Second`、`Kilobytes/Second`、`Megabytes/Second`、`Gigabytes/Second`、`Terabytes/Second`、`Bits/Second`、`Kilobits/Second`、`Megabits/Second`、`Gigabits/Second`、`Terabits/Second`、`Count/Second`、`None`

デフォルトは `None` です。

> **Note**
>
> API に適用される CloudWatch `PutMetricData`すべてのクォータは、このコンポーネントでパブリッシュするメトリクスに適用されます。以下の制限は特に重要です。
>
> - API ペイロードに適用される 40 KB の制限

- API リクエストごとに 20 個のメトリクス
- `PutMetricData` API の 150 トランザクション/秒 (TPS)

詳細については、「 CloudWatch ユーザーガイド」のCloudWatch 「 サービスクォータ」を参照してください。

Example 入力例

```
{
  "request": {
    "namespace": "Greengrass",
    "metricData": {
      "metricName": "latency",
      "dimensions": [
        {
          "name": "hostname",
          "value": "test_hostname"
        }
      ],
      "timestamp": 1539027324,
      "value": 123.0,
      "unit": "Seconds"
    }
  }
}
```

出力データ

このコンポーネントは、デフォルトで次のローカルパブリッシュ/サブスクライブトピックに出力データとしてレスポンスをパブリッシュします。カスタムコンポーネントでこのトピックに関するメッセージへサブスクライブする方法の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。

オプションで、`PubSubToIoTCore` 設定パラメータを `true` に設定することで。このコンポーネントが MQTT トピックにパブリッシュするように設定することができます。カスタムコンポーネントで MQTT トピックのメッセージにサブスクライブする方法の詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。

> **Note**
>
> コンポーネントバージョン 2.0.x では、デフォルトでレスポンスが MQTT トピックに出力データとしてパブリッシュされます。このトピックは、レガシーサブスクリプションルーターコンポーネントの設定で `subject` として指定する必要があります。

既定のトピック: `cloudwatch/metric/put/status`

Example 出力例: 成功

レスポンスには、メトリクスデータの名前空間と CloudWatch レスポンスの `RequestId`フィールドが含まれます。

```
{
  "response": {
    "cloudwatch_rid": "70573243-d723-11e8-b095-75ff2EXAMPLE",
    "namespace": "Greengrass",
    "status": "success"
  }
}
```

Example 出力例: 失敗

```
{
  "response" : {
    "namespace": "Greengrass",
    "error": "InvalidInputException",
    "error_message": "cw metric is invalid",
    "status": "fail"
  }
}
```

> **Note**
>
> コンポーネントが、接続エラーなどの再試行可能なエラーを検出した場合、次のバッチでパブリッシュを再試行します。

ライセンス

このコンポーネントには、次のサードパーティーソフトウェア/ライセンス品が含まれています。

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF ライセンス
- docutils/BSD ライセンス、GNU 一般パブリックライセンス (GPL)、Python Software Foundation ライセンス、パブリックドメイン
- jmespath/MIT ライセンス
- s3transfer/Apache License 2.0
- urllib3/MIT ライセンス

このコンポーネントは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされています。

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/aws.greengrass.Cloudwatch.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.Cloudwatch.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* を AWS IoT Greengrass ルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.Cloudwatch.log
  ```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\aws.greengrass.Cloudwatch.log -Tail 10 -Wait
```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

v3.x

バージョン	変更
3.1.0	バグ修正と機能向上 • HTTPS ネットワークプロキシ設定へのサポートが追加されました。詳細については、ポート 443 での接続またはネットワークプロキシを通じた接続およびコアデバイスが HTTPS プロキシを信頼できるようにするを参照してください。
3.0.0	このバージョンの CloudWatch メトリクスコンポーネントでは、バージョン 2.x とは異なる設定パラメータが必要です。バージョン 2.x でデフォルト以外の設定を使用し、v2.x から v3.x にアップグレードする場合は、コンポーネントの設定を更新する必要があります。詳細については、「CloudWatchメトリクスコンポーネントの設定」を参照してください。 新機能 • Windows を実行するコアデバイスのサポートが追加されました。 • コンポーネントタイプを Lambda コンポーネントから汎用コンポーネントに変更しました。このコンポーネントは、サブスクリプションを作成するのにレガシーサブスクリプションルーターコンポーネントに依存しなくなりました。 • コンポーネントがメッセージを受信するためにサブスクライブするトピックを指定するための、新しい `InputTopic` 設定パラメータが追加されました。 • コンポーネントがステータスレスポンスをパブリッシュするトピックを指定するための、新しい `OutputTopic` 設定パラメータが追加されました。

バージョン	変更
	• AWS IoT Core MQTT トピックを発行およびサブスクライブするかどうかを指定する新しいPubSubToIoTCore 設定パラメータを追加しました。 • コンポーネントの依存関係をインストールするインストールスクリプトをオプションで無効にできる、新しい UseInstaller 設定パラメータが追加されました。 バグ修正と機能向上 入力データでの重複するタイムスタンプへのサポートが追加されました。

v2.x

バージョン	変更
2.1.3	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.1.1	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.0	新機能 • HTTPS ネットワークプロキシ設定へのサポートが追加されました。詳細については、ポート 443 での接続またはネットワークプロキシを通じた接続およびコアデバイスが HTTPS プロキシを信頼できるようにするを参照してください。
2.0.8	バグ修正と機能向上 • 入力データでの重複するタイムスタンプへのサポートが追加されました。

バージョン	変更
	• Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.0.7	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.0.6	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.0.5	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.0.4	Greengrass nucleus バージョン 2.1.0 のリリース用にバージョンが更新されました。
2.0.3	当初のバージョン

以下も参照してください。

- 「Amazon CloudWatch ユーザーガイド」の「Amazon メトリクスの使用 CloudWatch 」
- PutMetricData 「Amazon CloudWatch API リファレンス」の「」

AWS IoT Device Defender

AWS IoT Device Defender コンポーネント (`aws.greengrass.DeviceDefender`) は、Greengrass コアデバイスの状態の変化を管理者に通知します。これは、侵害されたデバイスを示す可能性のある異常な動作を特定するのに役立ちます。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT Device Defender」を参照してください。

このコンポーネントは、コアデバイスのシステムメトリクスを読み込みます。その後、そのメトリクスを AWS IoT Device Defender にパブリッシュします。このコンポーネントが報告するメトリクスを読み取り、解釈する方法の詳細については、「AWS IoT Core デベロッパーガイド」の「デバイスメトリクスドキュメントの仕様」を参照してください。

> **Note**
>
> このコンポーネントは、 の Device Defender コネクタと同様の機能を提供します AWS IoT Greengrass V1。詳細については、「AWS IoT Greengrass V1 デベロッパーガイド」の「Device Defender コネクタ」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 3.1.x
- 3.0.x
- 2.0.x

コンポーネントの各バージョンに対する変更については、「変更ログ」を参照してください。

タイプ

v3.x

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

v2.x

このコンポーネントは Lambda コンポーネントです (`aws.greengrass.lambda`)。Greengrass nucleus は、Lambda ランチャーコンポーネントを使用してこのコンポーネントの Lambda 関数を実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

v3.x

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

v2.x

このコンポーネントは、Linux コアデバイスにのみインストールできます。

要件

このコンポーネントには次の要件があります。

v3.x

- Python バージョン 3.7 がコアデバイスにインストールされ、PATH 環境変数に追加されています。
- AWS IoT Device Defender Detect 機能を使用して違反をモニタリングするように が設定されました。詳細については、「AWS IoT Core デベロッパーガイド」の「検出」を参照してください。

v2.x

- コアデバイスは、Lambda 関数を実行するための要件を満たしている必要があります。コアデバイスが、コンテナ化された Lambda 関数を実行させる場合、そのデバイスは要件を満たす必要があります。詳細については、「Lambda 関数の要件」を参照してください。
- Python バージョン 3.7 がコアデバイスにインストールされ、PATH 環境変数に追加されています。
- AWS IoT Device Defender Detect 機能を使用して違反をモニタリングするように が設定されました。詳細については、「AWS IoT Core デベロッパーガイド」の「検出」を参照してください。
- コアデバイスにインストールされた psutil ライブラリ。このコンポーネントの動作が確認された最新のバージョンはバージョン 5.7.0 です。
- コアデバイスにインストールされた cbor ライブラリ。このコンポーネントの動作が確認された最新のバージョンはバージョン 1.0.0 です。
- このコンポーネントから出力データを受信するには、このコンポーネントをデプロイするときに、次の設定更新プログラムをレガシーサブスクリプションルーターのコンポーネント (`aws.greengrass.LegacySubscriptionRouter`) のためにマージする必要があります。この設定は、このコンポーネントがレスポンスを公開するトピックを指定します。

  Legacy subscription router v2.1.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-device-defender": {
        "id": "aws-greengrass-device-defender",
        "source": "component:aws.greengrass.DeviceDefender",
        "subject": "$aws/things/+/defender/metrics/json",
        "target": "cloud"
      }
    }
  }
  ```

  Legacy subscription router v2.0.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-device-defender": {
        "id": "aws-greengrass-device-defender",
  ```

```
      "source": "arn:aws:lambda:region:aws:function:aws-greengrass-device-
defender:version",
      "subject": "$aws/things/+/defender/metrics/json",
      "target": "cloud"
    }
  }
}
```

- *region* AWS リージョン は、使用する に置き換えます。
- *#####*を、このコンポーネントが実行する Lambda 関数のバージョンに置き換えます。Lambda 関数のバージョンを確認するには、デプロイするこのコンポーネントのバージョンの recipe を確認する必要があります。AWS IoT Greengrass コンソールで、このコンポーネントの詳細ページを開き、[Lambda function] (Lambda 関数) の key-value ペアを見つけます。このキー値のペアには、Lambda 関数の名前とバージョンが含まれます。

> **Important**
>
> このコンポーネントをデプロイするたびに、レガシーサブスクリプションルーターの Lambda 関数のバージョンを更新する必要があります。これにより、デプロイするコンポーネントバージョンに正しい Lambda 関数のバージョンが使用されることが保証されます。

詳細については、「デプロイの作成」を参照してください。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

3.1.1

次の表に、このコンポーネントのバージョン 3.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <3.0.0	ソフト
トークン交換サービス	>=0.0.0	ハード

3.0.0 - 3.0.2

次の表に、このコンポーネントのバージョン 3.0.0 から 3.0.2 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <3.0.0	ソフト
トークン交換サービス	>=0.0.0	ハード

2.0.10 and 2.0.11

次の表に、このコンポーネントのバージョン 2.0.10 および 2.0.11 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.9

次の表に、このコンポーネントのバージョン 2.0.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ハード

依存関係	互換性のあるバージョン	依存関係タイプ
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.8

次の表に、このコンポーネントのバージョン 2.0.8 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.7

次の表に、このコンポーネントのバージョン 2.0.7 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.6

次の表に、このコンポーネントのバージョン 2.0.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.5

次の表に、このコンポーネントのバージョン 2.0.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.4

次の表に、このコンポーネントのバージョン 2.0.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.3

次の表に、このコンポーネントのバージョン 2.0.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.3 <2.1.0	ハード
Lambda ランチャー	>=1.0.0	ハード
Lambda ランタイム	>=1.0.0	ソフト
トークン交換サービス	>=1.0.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

v3.x

`PublishRetryCount`

パブリッシュが再試行される回数。この機能は、バージョン 3.1.1 で使用できます。

最小値は 0 です。

最大数は 72 です。

デフォルト: 5

`SampleIntervalSeconds`

(オプション) コンポーネントがメトリクスを収集してレポートする各サイクルの間隔 (秒単位)。

最小値は 300 秒 (5 分) です。

デフォルト: 300 秒

`UseInstaller`

(オプション) このコンポーネントの依存関係をインストールするために、このコンポーネントでインストーラスクリプトを使用するかどうかを定義するブール値。

依存関係のインストールにカスタムスクリプトを使用する場合、またはビルド済みの Linux イメージにランタイムの依存関係を含める場合は、この値を `false` に設定します。このコンポーネントを使用するには、依存関係を含む次のライブラリをインストールし、デフォルトの Greengrass システムユーザーを利用できるようにする必要があります。

- AWS IoT Device SDK v2 for Python
- cbor ライブラリ。このコンポーネントの動作が確認された最新のバージョンはバージョン 1.0.0 です。
- psutil ライブラリ。このコンポーネントの動作が確認された最新のバージョンはバージョン 5.7.0 です。

> **Note**
>
> HTTPS プロキシを使用するように設定したコアデバイスで、このコンポーネントのバージョン 3.0.0 または 3.0.1 を使用する場合は、この値を `false` に設定する必要があります。インストーラスクリプトは、このコンポーネントのこれらのバージョンでは HTTPS プロキシの背後からの操作をサポートしていません。

デフォルト: `true`

v2.x

> **Note**
>
> このコンポーネントのデフォルト設定には、Lambda 関数のパラメータが含まれます。デバイスにこのコンポーネントを設定するには、次のパラメータのみを編集することをお勧めします。

`lambdaParams`

このコンポーネントの Lambda 関数のパラメータを含むオブジェクト。このオブジェクトには、次の情報が含まれます。

`EnvironmentVariables`

Lambda 関数のパラメータを含むオブジェクト。このオブジェクトには、次の情報が含まれます。

`PROCFS_PATH`

(オプション) `/proc` フォルダへのパス。

- コンテナでこのコンポーネントを実行するには、デフォルト値の `/host-proc` を使用します。このコンポーネントは、デフォルトでコンテナ内で実行されます。
- このコンポーネントをコンテナなしモードで実行するには、このパラメータに `/proc` を指定します。

デフォルト: `/host-proc`。これは、このコンポーネントがコンテナ内に `/proc` フォルダをマウントするためのデフォルトのパスです。

> **Note**
>
> このコンポーネントは、このフォルダに対して読み取り専用のアクセス許可を有します。

`SAMPLE_INTERVAL_SECONDS`

(オプション) コンポーネントがメトリクスを収集してレポートする各サイクルの間隔 (秒単位)。

最小値は 300 秒 (5 分) です。

デフォルト: 300 秒

`containerMode`

(オプション) このコンポーネントのコンテナ化モード。次のオプションから選択します。

- `GreengrassContainer` – コンポーネントは、 AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。
- `NoContainer` - コンポーネントは、分離されたランタイム環境では実行されません。

  このオプションを指定する場合は、`PROCFS_PATH` 環境変数パラメータに `/proc` を指定する必要があります。

デフォルト: `GreengrassContainer`

`containerParams`

(オプション) このコンポーネントのコンテナパラメータを含むオブジェクト。`containerMode` の `GreengrassContainer` を指定した場合、コンポーネントはこれらのパラメータを使用します。

このオブジェクトには、次の情報が含まれます。

`memorySize`

(オプション) コンポーネントに割り当てるメモリ量 (KB 単位)。

デフォルトは 50,000 KB です。

`pubsubTopics`

(オプション) コンポーネントがメッセージを受信するためにサブスクライブするトピックを含むオブジェクト。各トピックと、コンポーネントが から MQTT トピックをサブスクライブするか、ローカルのパブリッシュ/サブスクライブトピックをサブスクライブ AWS IoT Core するかを指定できます。

このオブジェクトには、次の情報が含まれます。

`0` - これは文字列としての配列インデックスです。

次の情報が含まれるオブジェクト。

`type`

(オプション) このコンポーネントがメッセージをサブスクライブするために使用するパブリッシュ/サブスクライブメッセージングのタイプ。次のオプションから選択します。

- `PUB_SUB` - ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブします。このオプションを選択した場合、トピックに MQTT ワイルドカードを含めることはできません。このオプションを指定したときに、カスタムコンポーネントからメッセージを送信する方法の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。
- `IOT_CORE` – AWS IoT Core MQTT メッセージをサブスクライブします。このオプションを選択した場合、トピックに MQTT ワイルドカードを含めることができます。このオプションを指定したときに、カスタムコンポーネントからメッセージを送信する方法の詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。

デフォルト: PUB_SUB

`topic`

(オプション) コンポーネントがメッセージを受信するためにサブスクライブするトピック。type の IotCore を指定した場合、このトピックで MQTT ワイルドカード (+ および #) を使用できます。

Example 例: 設定マージの更新 (コンテナモード)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "PROCFS_PATH": "/host_proc"
    }
  },
  "containerMode": "GreengrassContainer"
}
```

Example 例: 設定マージの更新 (コンテナモードなし)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "PROCFS_PATH": "/proc"
    }
  },
  "containerMode": "NoContainer"
}
```

入力データ

このコンポーネントは、メッセージを入力データとして受け入れません。

出力データ

このコンポーネントは、 のセキュリティメトリクスを次の予約済みトピックに発行します AWS IoT Device Defender。このコンポーネントは、メトリクスを発行するときに、 をコアデバイスの名前*coreDeviceName*に置き換えます。

トピック (AWS IoT Core MQTT): $aws/things/*coreDeviceName*/defender/metrics/json

Example 出力例


```
{
  "header": {
    "report_id": 1529963534,
    "version": "1.0"
  },
  "metrics": {
    "listening_tcp_ports": {
      "ports": [
        {
          "interface": "eth0",
          "port": 24800
        },
        {
          "interface": "eth0",
          "port": 22
        },
        {
          "interface": "eth0",
          "port": 53
        }
      ],
      "total": 3
    },
    "listening_udp_ports": {
      "ports": [
        {
          "interface": "eth0",
          "port": 5353
        },
        {
          "interface": "eth0",
          "port": 67
        }
      ],
      "total": 2
    },
    "network_stats": {
      "bytes_in": 1157864729406,
      "bytes_out": 1170821865,
      "packets_in": 693092175031,
      "packets_out": 738917180
    },
```

```
    "tcp_connections": {
      "established_connections":{
        "connections": [
          {
            "local_interface": "eth0",
            "local_port": 80,
            "remote_addr": "192.168.0.1:8000"
          },
          {
            "local_interface": "eth0",
            "local_port": 80,
            "remote_addr": "192.168.0.1:8000"
          }
        ],
        "total": 2
      }
    }
  }
}
```

このコンポーネントが報告するメトリクスの詳細については、「AWS IoT Core デベロッパーガイド」の「デバイスメトリクスドキュメントの仕様」を参照してください。

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/aws.greengrass.DeviceDefender.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.DeviceDefender.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* を AWS IoT Greengrass ルートフォルダへのパスに置き換えます。

Linux

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.DeviceDefender.log
```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\aws.greengrass.DeviceDefender.log -Tail 10 -
Wait
```

ライセンス

このコンポーネントは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされています。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

v3.x

バージョン	変更
3.1.1	バグ修正と機能向上 • ネットワークの停止後に接続の回復に失敗した場合に、クライアント接続の再試行を追加します。 • メトリクスを公開するための構成可能な再試行を追加します。
3.1.0	バグ修正と機能向上 • HTTPS ネットワークプロキシ設定へのサポートが追加されました。詳細については、ポート 443 での接続またはネットワークプロキシを通じた接続およびコアデバイスが HTTPS プロキシを信頼できるようにするを参照してください。
3.0.1	コンポーネントがメトリクスのデルタ値を計算する方法で生じる問題を修正しました。

バージョン	変更
3.0.0	**Warning** このバージョンは現在利用できません。このバージョンの改善は、このコンポーネントのそれ以降のバージョンで利用できます。 当初のバージョン

v2.x

バージョン	変更
2.0.11	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.0.10	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.0.9	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.0.8	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.0.7	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.0.6	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.0.5	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.0.4	Greengrass nucleus バージョン 2.1.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.0.3	当初のバージョン

ディスクスプーラ

ディスクスプーラコンポーネント (`aws.greengrass.DiskSpooler`) は、Greengrass コアデバイスから にスプールされたメッセージ用の永続的ストレージオプションを提供します AWS IoT Core。このコンポーネントは、これらの送信メッセージをディスクに保存します。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 1.0.x

タイプ

このコンポーネントはプラグインコンポーネント (`aws.greengrass.plugin`) です。Greengrass nucleus は、nucleus と同じ Java バーチャルマシン (JVM) でこのコンポーネントを実行します。コアデバイスでこのコンポーネントのバージョンを変更するとき、nucleus が再起動します。

このコンポーネントは、Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- `storageType` を `Disk` に設定してこのコンポーネントを使用する必要があります。これは [Greengrass nucleus 設定] で設定できます。
- `maxSizeInBytes` はデバイス上の空きスペースを超える設定にしないでください。これは [Greengrass nucleus 設定] で設定できます。
- ディスクスプーラコンポーネントは、VPC での実行がサポートされています。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

1.0.1 – 1.0.3

次の表に、このコンポーネントのバージョン 1.0.1 から 1.0.3 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.11.0 <2.13.0	ハード

1.0.0

次の表に、このコンポーネントのバージョン 1.0.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.11.0 <2.12.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

使用方法

ディスクスプーラコンポーネントを使用するには、`aws.greengrass.DiskSpooler` をデプロイする必要があります。

このコンポーネントを設定して使用するには、`pluginName` を `aws.greengrass.DiskSpooler` に設定する必要があります。

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* を AWS IoT Greengrass ルートフォルダへのパスに置き換えます。

Linux

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
1.0.3	バグ修正と機能向上 データベース接続を再利用することでパフォーマンスが向上します。
1.0.2	バグ修正と機能向上 特定のケースで MQTT メッセージ形式フィールドが保持されない問題を修正しました。
1.0.1	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
1.0.0	当初のバージョン

Docker アプリケーションマネージャー

Docker アプリケーションマネージャーコンポーネント (`aws.greengrass.DockerApplicationManager`) を使用することで、AWS IoT Greengrass は Amazon Elastic Container Registry (Amazon ECR) でホストされているパブリックイメージレジストリとプライベートレジストリから Docker イメージをダウンロードできるようになります。AWS IoT Greengrass が Amazon ECR のプライベートリポジトリからイメージをダウンロードするための認証情報を可能になります。

Docker コンテナを実行するカスタムコンポーネントを開発するときには、Docker アプリケーションマネージャーを依存関係として含めて、コンポーネント内のアーティファクトとして指定された Docker イメージをダウンロードします。詳細については、「Docker コンテナの実行」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Docker Engine 1.9.1 以降が Greengrass コアにインストールされていいること。バージョン 20.10 は、AWS IoT Greengrass Core ソフトウェアとの動作が確認されている最新バージョンです。Docker コンテナを実行するコンポーネントをデプロイする前に、コアデバイスに直接、Docker をインストールしておく必要があります。
- このコンポーネントをデプロイする前に、Docker デーモンがコアデバイス上で起動し、実行されています。
- Docker イメージは、次のいずれかのサポートされているイメージソースに格納されています。
  - Amazon Elastic Container Registry (Amazon ECR) のパブリックイメージリポジトリおよびプライベートイメージリポジトリ
  - パブリック Docker Hub リポジトリ
  - パブリック Docker の信頼レジストリ
- カスタム Docker コンテナコンポーネントにアーティファクトとして含まれる Docker イメージ。次の URI 形式を使用して、Docker イメージを指定します。
  - プライベート Amazon ECR イメージ: `docker:`*`account-id`*`.dkr.ecr.`*`region`*`.amazonaws.com/`*`repository`*`/`*`image`*`[`*`:tag|@digest`*`]`
  - パブリック Amazon ECR イメージ: `docker:public.ecr.aws/`*`repository`*`/`*`image`*`[`*`:tag|@digest`*`]`
  - パブリック Docker Hub イメージ: `docker:`*`name`*`[`*`:tag|@digest`*`]`

  詳細については、「Docker コンテナの実行」を参照してください。

  > **Note**
  >
  > イメージのアーティファクト URI にイメージタグまたはイメージダイジェストを指定しなかった場合、Docker アプリケーションマネージャーがカスタム Docker コンテナコンポーネントをデプロイするときに、そのイメージの最新バージョンを取得します。すべてのコアデバイスで、確実に同じバージョンのイメージが実行されるようにするため、アーティファクト URI にイメージタグまたはイメージダイジェストを含めることをお勧めします。
- Docker コンテナコンポーネントを実行するシステムユーザーには、ルート権限または管理者権限が必要です。権限がない場合は、ルート権限または管理者権限を持たないユーザーとして実行されるように Docker を設定する必要があります。

- Linux デバイスでは、ユーザーを docker グループに追加することで、sudo のない docker コマンドを呼び出せます。
- Windows デバイスでは、ユーザーを docker-users グループ に追加することで、管理者の権限のない docker コマンドを呼び出せます。

Linux or Unix

Docker コンテナコンポーネントの実行に使用する ggc_user または非ルートユーザーを docker グループに追加するには、次のコマンドを実行します。

```
sudo usermod -aG docker ggc_user
```

詳細については、「Docker を非ルートユーザーとして管理する」を参照してください。

Windows Command Prompt (CMD)

Docker コンテナコンポーネントの実行に使用する ggc_user またはユーザーを docker-users グループに追加するには、次のコマンドを管理者として実行します。

```
net localgroup docker-users ggc_user /add
```

Windows PowerShell

Docker コンテナコンポーネントの実行に使用する ggc_user またはユーザーを docker-users グループに追加するには、次のコマンドを管理者として実行します。

```
Add-LocalGroupMember -Group docker-users -Member ggc_user
```

- AWS IoT Greengrass Core ソフトウェアがネットワークプロキシを使用するように設定している場合、Docker が同じプロキシサーバーを使用するように設定する必要があります。
- Docker イメージが Amazon ECR プライベートレジストリに格納されている場合は、トークン交換サービスコンポーネントを依存関係として Docker コンテナコンポーネントに含める必要があります。また、Greengrass デバイスのロールは、以下の IAM ポリシー例で示されているように、ecr:GetAuthorizationToken、ecr:BatchGetImage、ecr:GetDownloadUrlForLayer アクションを許可する必要があります。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
```

```
      "Action": [
        "ecr:GetAuthorizationToken",
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

- Docker アプリケーションマネージャーコンポーネントは、VPC での実行がサポートされています。このコンポーネントを VPC にデプロイするには、以下が必要です。
  - Docker アプリケーションマネージャーコンポーネントには、イメージをダウンロードするための接続が必要です。例えば、ECR を使用する場合は、次のエンドポイントへの接続が必要です。
    - `*.dkr.ecr.region.amazonaws.com` (VPC エンドポイント `com.amazonaws.region.ecr.dkr`)
    - `api.ecr.region.amazonaws.com` (VPC エンドポイント `com.amazonaws.region.ecr.api`)

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
`ecr.region.amazonaws.com`	443	いいえ	Amazon ECR から Docker イメージをダウンロードする場合に必要です。

エンドポイント	ポート	必要	説明

`hub.docker.com`
 	
`registry.hub.docker.com/v1`
	443	いいえ	Docker Hub から Docker イメージをダウンロードする場合に必要です。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.0.11

次の表に、このコンポーネントのバージョン 2.0.11 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.13.0	ソフト

2.0.10

次の表に、このコンポーネントのバージョン 2.0.10 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.12.0	ソフト

2.0.9

次の表に、このコンポーネントのバージョン 2.0.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.11.0	ソフト

2.0.8

次の表に、このコンポーネントのバージョン 2.0.8 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.10.0	ソフト

2.0.7

次の表に、このコンポーネントのバージョン 2.0.7 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.9.0	ソフト

2.0.6

次の表に、このコンポーネントのバージョン 2.0.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.8.0	ソフト

2.0.5

次の表に、このコンポーネントのバージョン 2.0.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.7.0	ソフト

2.0.4

次の表に、このコンポーネントのバージョン 2.0.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.6.0	ソフト

2.0.3

次の表に、このコンポーネントのバージョン 2.0.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.5.0	ソフト

2.0.2

次の表に、このコンポーネントのバージョン 2.0.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.4.0	ソフト

2.0.1

次の表に、このコンポーネントのバージョン 2.0.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.3.0	ソフト

2.0.0

次の表に、このコンポーネントのバージョン 2.0.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.2.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントに設定パラメータはありません。

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.0.11	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.0.10	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.0.9	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.0.8	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.0.7	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.0.6	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.0.5	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.0.4	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.0.3	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.0.2	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.0.1	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.0.0	当初のバージョン

以下も参照してください。

- Docker コンテナの実行

Kinesis Video Streams 向けのエッジコネクタ

Kinesis Video Streams コンポーネントのエッジコネクタ (`aws.iot.EdgeConnectorForKVS`) は、ローカルカメラからビデオフィードを読み取り、ストリームを Kinesis Video Streams に発行します。このコンポーネントは、リアルタイムストリーミングプロトコル (RTSP) を使用してインターネットプロトコル (IP) カメラからビデオフィードを読み取るように設定できます。次に、Amazon Managed Grafana またはローカルの Grafana サーバーでダッシュボードを設定すると、ビデオストリームの監視とやり取りを行えます。

このコンポーネントを と統合AWS IoT TwinMaker して、Grafana ダッシュボードでビデオストリームを表示および制御できます。AWS IoT TwinMaker は、物理システムの運用デジタルツインを構築できる AWSのサービスです。AWS IoT TwinMaker を使用して、センサー、カメラ、エンタープライズアプリケーションのデータを視覚化し、物理的な工場、建物、または産業工場を追跡できます。また、このデータを使用してオペレーションの監視、エラーの診断、エラーの修復を行うことができます。詳細については、『AWS IoT TwinMaker ユーザーガイド』の「What is AWS IoT TwinMaker ? (とは ?)」 を参照してください。

このコンポーネントはその設定を、産業データをモデル化して保存する AWS サービスである AWS IoT SiteWise に保存します。AWS IoT SiteWise では、アセットとはデバイス、機器、またはその他のオブジェクトのグループなどのオブジェクトを表します。このコンポーネントを設定して使用するには、各 Greengrass コアデバイスと、各コアデバイスに接続された各 IP カメラに、AWS IoT SiteWise アセットを作成します。各アセットにはライブストリーミング、オンデマンドアップロード、ローカルキャッシュなどの機能を制御するために設定を行うプロパティがあります。各カメラ

の URL を指定するには、カメラの URL が含まれているシークレットを AWS Secrets Manager に作成します。カメラが認証を必要とする場合は、URL にユーザー名とパスワードも指定します。次に、IP カメラのアセットプロパティでそのシークレットを指定します。

このコンポーネントは、各カメラのビデオストリームを Kinesis のビデオストリームにアップロードします。送信先の Kinesis のビデオストリームの名前は、各カメラの AWS IoT SiteWise アセット設定で指定します。Kinesisの ビデオストリームが存在しない場合、このコンポーネントはユーザーのためにそれを作成します。

AWS IoT TwinMaker には、これらのAWS IoT SiteWiseアセットと Secrets Manager シークレットを作成するために実行できるスクリプトが用意されています。これらのリソースを作成する方法、およびこのコンポーネントをインストール、設定、使用する方法の詳細については、「 AWS IoT TwinMaker ユーザーガイド」の「 AWS IoT TwinMaker ビデオ統合」を参照してください。

> **Note**
>
> Kinesis Video Streams コンポーネントのエッジコネクタは、以下の AWS リージョン でのみ利用できます。
>
> - 米国東部 (バージニア北部)
> - 米国西部 (オレゴン)
> - 欧州 (フランクフルト)
> - 欧州 (アイルランド)
> - アジアパシフィック (シンガポール)

トピック

- ローカルログファイル
- 変更ログ
- 以下も参照してください。

バージョン

このコンポーネントには、次のバージョンがあります。

- 1.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、Linux コアデバイスにのみインストールできます。

要件

このコンポーネントには次の要件があります。

- コンポーネント設定はコアデバイスごとに一意である必要があるため、このコンポーネントをデプロイできるのはシングルコアデバイスに対してのみです。このコンポーネントをコアデバイスのグループにデプロイすることはできません。
- コアデバイスに GStreamer 1.18.4 以降がインストールされています。詳細については、「GStreamer のインストール」を参照してください。

  `apt` を使用するデバイスでは、以下のコマンドを実行して GStreamer をインストールできます。

```
sudo apt install -y libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev
 gstreamer1.0-plugins-base-apps
sudo apt install -y gstreamer1.0-libav
sudo apt install -y gstreamer1.0-plugins-bad gstreamer1.0-plugins-good gstreamer1.0-
plugins-ugly gstreamer1.0-tools
```

- 各コアデバイス用 AWS IoT SiteWise アセット。この AWS IoT SiteWise アセットはコアデバイスを表します。このアセットの作成方法の詳細については、「 ユーザーガイド」の「 AWS IoT TwinMaker ビデオ統合」を参照してください。 AWS IoT TwinMaker
- 各コアデバイスに接続する各 IP カメラ用の AWS IoT SiteWise アセット。これらの AWS IoT SiteWise アセットは、各コアデバイスにビデオをストリーミングするカメラを表します。各カメラのアセットを、カメラに接続するコアデバイスのアセットに関連付ける必要があります。カメラアセットには、Kinesis ビデオストリーム、認証シークレット、およびビデオストリーミングパラメータを指定する設定を行えるプロパティがあります。カメラアセットを作成および設定する方法の詳細については、「 ユーザーガイド」のAWS IoT TwinMaker 「ビデオ統合」を参照してください。 AWS IoT TwinMaker
- 各 IP カメラの AWS Secrets Manager シークレット。このシークレットは、キーと値のペアを定義する必要があります。この場合、キーは `RTSPStreamUrl`、値はカメラの URLになります。カメラが認証を必要とする場合は、この URL にユーザー名とパスワードを含めます。このコンポーネントに必要なリソースを作成するとき、スクリプトを使用してシークレットを作成できます。詳細については、「 ユーザーガイド」の「 AWS IoT TwinMaker ビデオ統合」を参照してください。 AWS IoT TwinMaker

  Secrets Manager コンソールと API を使用して、追加のシークレットを作成することもできます。詳細については、「AWS Secrets Manager ユーザーガイド」の「シークレットの作成」を参照してください。
- 次の IAM ポリシーの例で示されているように、Greengrass トークン交換ロールは、以下の AWS Secrets Manager、AWS IoT SiteWise、および Kinesis Video Streams のアクションを許可する必要があります。

> **Note**
>
> このポリシーの例は、デバイスが **`IPCamera1Url`** と **`IPCamera2Url`** という名前のシークレットの値を取得することを許可します。各 IP カメラを設定するときは、そのカメラの URL を含むシークレットを指定します。カメラが認証を必要とする場合は、URL にユーザー名とパスワードも指定します。コアデバイスのトークン交換ロールは、各 IP カメラが接続するために、シークレットへのアクセスを許可する必要があります。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
```

```
      "Action": [
        "secretsmanager:GetSecretValue"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:secretsmanager:region:account-id:secret:IPCamera1Url",
        "arn:aws:secretsmanager:region:account-id:secret:IPCamera2Url"
      ]
    },
    {
      "Action": [
        "iotsitewise:BatchPutAssetPropertyValue",
        "iotsitewise:DescribeAsset",
        "iotsitewise:DescribeAssetModel",
        "iotsitewise:DescribeAssetProperty",
        "iotsitewise:GetAssetPropertyValue",
        "iotsitewise:ListAssetRelationships",
        "iotsitewise:ListAssets",
        "iotsitewise:ListAssociatedAssets",
        "kinesisvideo:CreateStream",
        "kinesisvideo:DescribeStream",
        "kinesisvideo:GetDataEndpoint",
        "kinesisvideo:PutMedia",
        "kinesisvideo:TagStream"
      ],
      "Effect": "Allow",
      "Resource": [
        "*"
      ]
    }
  ]
}
```

> **Note**
>
> カスタマー管理の AWS Key Management Service キーを使用してシークレットを暗号化する場合は、デバイスのロールで `kms:Decrypt` アクションも許可する必要があります。

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
kinesisvideo. *region*.amazonaws.com	443	はい	Kinesis Video Streams にデータをアップロードします。
data.iotsitewise. *region*.amazonaws.com	443	はい	ビデオストリームのメタデータを AWS IoT SiteWise に発行します。
secretsmanager. *region*.amazonaws.com	443	はい	コアデバイスにカメラ URL シークレットをダウンロードします。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの

依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

次の表に、このコンポーネントのバージョン 1.0.0 から 1.0.4 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
トークン交換サービス	>=2.0.3	ハード
ストリームマネージャー	>=2.0.9	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

`SiteWiseAssetIdForHub`

このコアデバイスを表す AWS IoT SiteWise アセットの ID。このアセットを作成し、それを使用してこのコンポーネントとやり取りする方法の詳細については、AWS IoT TwinMaker 「 ユーザーガイド」の「 AWS IoT TwinMaker ビデオ統合」を参照してください。

Example 例: 設定マージの更新

```
{
  "SiteWiseAssetIdForHub": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
}
```

ライセンス

このコンポーネントには、次のサードパーティーソフトウェア/ライセンス品が含まれています。

- Quartz Job Scheduler / Apache License 2.0
- Java bindings for GStreamer 1.x / GNU Lesser General Public License v3.0

使用方法

このコンポーネントを設定してやり取りするには、コアデバイスとその接続先の IP カメラを表す AWS IoT SiteWise アセットのプロパティを設定します。を通じて Grafana ダッシュボードでビデオストリームを視覚化して操作することもできますAWS IoT TwinMaker。詳細については、「 ユーザーガイド」の「 AWS IoT TwinMaker ビデオ統合」を参照してください。 AWS IoT TwinMaker

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

```
/greengrass/v2/logs/aws.iot.EdgeConnectorForKVS.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。をAWS IoT Greengrassルートフォルダへのパス*/greengrass/v2*に置き換えます。

  ```
  sudo tail -f /greengrass/v2/logs/aws.iot.EdgeConnectorForKVS.log
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
1.0.4	バグ修正と機能向上 • ライブアップロード停止の原因となる問題を修正しました。
1.0.3	一般的なバグ修正と機能強化。
1.0.1	一般的なバグ修正と機能強化。
1.0.0	当初のバージョン

以下も参照してください。

- 「AWS IoT TwinMaker ユーザーガイド」の「AWS IoT TwinMaker とは」
- AWS IoT TwinMaker ユーザーガイドの AWS IoT TwinMaker ビデオ統合
- 「AWS IoT SiteWise ユーザーガイド」の「AWS IoT SiteWise とは」
- 「AWS IoT SiteWise ユーザーガイド」の「属性値の更新」
- 「AWS Secrets Manager ユーザーガイド」の「AWS Secrets Manager とは」
- 「AWS Secrets Manager ユーザーガイド」の「シークレットの作成と管理」

Greengrass CLI

Greengrass CLI コンポーネント (`aws.greengrass.Cli`) は、コアデバイス上でコンポーネントをローカルで開発およびデバッグするために使用できるローカルコマンドラインインターフェイスを提供します。Greengrass CLI を使用すると、たとえば、ローカルデプロイを作成し、コアデバイスでコンポーネントを再起動できます。

このコンポーネントは、 AWS IoT Greengrass Core ソフトウェアのインストール時にインストールできます。詳細については、「チュートリアル: AWS IoT Greengrass V2 の開始方法」を参照してください。

> **Important**
>
> このコンポーネントは、本番環境ではなく、開発環境でのみで使用することをお勧めします。このコンポーネントは、通常、本番環境では必要とされない情報や操作へのアクセスを提供します。このコンポーネントを必要なコアデバイスにのみデプロイして、最小特権の原則に従います。

このコンポーネントをインストールしたら、以下のコマンドを実行してヘルプドキュメントを表示します。このコンポーネントがインストールされると、*/greengrass/v2*/bin フォルダに `greengrass-cli` へのシンボリックリンクが追加されます。このパスから Greengrass CLI を実行するか、絶対パスなしで `greengrass-cli` を実行する PATH 環境変数に追加することができます。

Linux or Unix

```
/greengrass/v2/bin/greengrass-cli help
```

Windows

```
C:\greengrass\v2\bin\greengrass-cli help
```

次のコマンドを使用すると、たとえば com.example.HelloWorld という名前のコンポーネントが再起動されます。

Linux or Unix

```
sudo /greengrass/v2/bin/greengrass-cli component restart --names
 "com.example.HelloWorld"
```

Windows

```
C:\greengrass\v2\bin\greengrass-cli component restart --names
 "com.example.HelloWorld"
```

詳細については、「Greengrass コマンドラインインターフェイス」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.12.x

- 2.11.x
- 2.10.x
- 2.9.x
- 2.8.x
- 2.7.x
- 2.6.x
- 2.5.x
- 2.4.x
- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

タイプ

このコンポーネントはプラグインコンポーネント (`aws.greengrass.plugin`) です。Greengrass nucleus は、nucleus と同じ Java バーチャルマシン (JVM) でこのコンポーネントを実行します。コアデバイスでこのコンポーネントのバージョンを変更するとき、nucleus が再起動します。

このコンポーネントは、Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- AWS IoT Greengrass Core ソフトウェアを操作するには、Greengrass CLI を使用する権限が必要です。Greengrass CLI を使用するには、次のいずれかを実行します。
  - AWS IoT Greengrass Core ソフトウェアを実行するシステムユーザーを使用します。
  - root 権限または管理者権限を持つユーザーを使用する。Linux コアデバイスでは、sudo を使用して root 権限を取得できます。
  - 設定をデプロイする際に、AuthorizedPosixGroups または AuthorizedWindowsGroups 設定パラメータで指定したグループのシステムユーザーを使用する。詳細については、「Greengrass CLI コンポーネント設定」を参照してください。
- Greengrass CLI コンポーネントは VPC での実行がサポートされています。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.12.0 and 2.12.1

次の表に、このコンポーネントのバージョン 2.12.0 および 2.12.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.12.0 <2.13.0	ソフト

2.11.0 – 2.11.3

次の表に、このコンポーネントのバージョン 2.11.0 から 2.11.3 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.11.0 <2.12.0	ソフト

2.10.0 – 2.10.3

次の表に、このコンポーネントのバージョン 2.10.0 から 2.10.3 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.11.0	ソフト

2.9.0 – 2.9.6

次の表に、このコンポーネントのバージョン 2.9.0 から 2.9.6 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.10.0	ソフト

2.8.0 – 2.8.1

次の表に、このコンポーネントのバージョン 2.8.0 および 2.8.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.9.0	ソフト

2.7.0

次の表に、このコンポーネントのバージョン 2.7.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.8.0	ソフト

2.6.0

次の表に、このコンポーネントのバージョン 2.6.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.7.0	ソフト

2.5.0 – 2.5.6

次の表に、このコンポーネントのバージョン 2.5.0 から 2.5.6 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.6.0	ソフト

2.4.0

次の表に、このコンポーネントのバージョン 2.4.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.5.0	ソフト

2.3.0

次の表に、このコンポーネントのバージョン 2.3.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.4.0	ソフト

2.2.0

次の表に、このコンポーネントのバージョン 2.2.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.3.0	ソフト

2.1.0

次の表に、このコンポーネントのバージョン 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.2.0	ソフト

2.0.x

次の表に、このコンポーネントのバージョン 2.0.x の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.1.0	ソフト

> **Note**
>
> Greengrass nucleus の最小互換バージョンは、Greengrass CLI コンポーネントのパッチバージョンに対応しています。

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

2.5.x

`AuthorizedPosixGroups`

(オプション) システムグループのカンマ区切りリストを含む文字列。これらのシステムグループが Greengrass CLI を使用して AWS IoT Greengrass Core ソフトウェアとやり取りすることを許可します。グループ名またはグループ ID を指定できます。例:`group1,1002,group3`

は 3 つのシステムグループ (group1、1002 および group3) に Greengrass CLI の使用を認可します。

承認するグループを指定しない場合は、ルートユーザー (sudo) または AWS IoT Greengrass Core ソフトウェアを実行するシステムユーザーとして Greengrass CLI を使用できます。

`AuthorizedWindowsGroups`

(オプション) システムグループのカンマ区切りリストを含む文字列。これらのシステムグループが Greengrass CLI を使用して AWS IoT Greengrass Core ソフトウェアとやり取りすることを許可します。グループ名またはグループ ID を指定できます。例:`group1,1002,group3` は 3 つのシステムグループ (group1、1002 および group3) に Greengrass CLI の使用を認可します。

承認するグループを指定しない場合、Greengrass CLI を管理者または AWS IoT Greengrass Core ソフトウェアを実行するシステムユーザーとして使用できます。

Example 例: 設定マージの更新

次の設定例では、3 つの POSIX システムグループ (group1、1002 および group3) と 2 つの Windows ユーザーグループ (Device Operators と QA Engineers) に Greengrass CLI の利用を認可するように指定しています。

```
{
  "AuthorizedPosixGroups": "group1,1002,group3",
  "AuthorizedWindowsGroups": "Device Operators,QA Engineers"
}
```

2.4.x - 2.0.x

`AuthorizedPosixGroups`

(オプション) システムグループのカンマ区切りリストを含む文字列。これらのシステムグループが Greengrass CLI を使用して AWS IoT Greengrass Core ソフトウェアとやり取りすることを許可します。グループ名またはグループ ID を指定できます。例:`group1,1002,group3` は 3 つのシステムグループ (group1、1002 および group3) に Greengrass CLI の使用を認可します。

承認するグループを指定しない場合は、ルートユーザー (sudo) または AWS IoT Greengrass Core ソフトウェアを実行するシステムユーザーとして Greengrass CLI を使用できます。

Example 例: 設定マージの更新

次の設定例では、3 つのシステムグループ (group1、1002 および group3) に Greengrass CLI の利用を認可するように指定しています。

```
{
  "AuthorizedPosixGroups": "group1,1002,group3"
}
```

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* を AWS IoT Greengrass ルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.12.1	Greengrass nucleus バージョン 2.12.1 リリース用にバージョンが更新されました。
2.12.0	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.11.3	Greengrass nucleus バージョン 2.11.3 リリース用にバージョンが更新されました。
2.11.2	Greengrass nucleus バージョン 2.11.2 のリリース用にバージョンが更新されました。
2.11.1	Greengrass nucleus バージョン 2.11.1 のリリース用にバージョンが更新されました。
2.11.0	新機能 • ローカルのデプロイをキャンセルできます。 • ローカルのデプロイの障害処理ポリシーを設定できます。 • 詳細なデプロイステータスのレポートを改善しています。
2.10.3	Greengrass nucleus バージョン 2.10.3 のリリース用にバージョンが更新されました。
2.10.2	Greengrass nucleus バージョン 2.10.2 のリリース用にバージョンが更新されました。
2.10.1	Greengrass nucleus バージョン 2.10.1 のリリース用にバージョンが更新されました。
2.10.0	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.9.6	Greengrass nucleus バージョン 2.9.6 のリリース用にバージョンが更新されました。
2.9.5	Greengrass nucleus バージョン 2.9.5 のリリース用にバージョンが更新されました。

バージョン	変更
2.9.4	Greengrass nucleus バージョン 2.9.4 のリリース用にバージョンが更新されました。
2.9.3	Greengrass nucleus バージョン 2.9.3 のリリース用にバージョンが更新されました。
2.9.2	Greengrass nucleus バージョン 2.9.2 のリリース用にバージョンが更新されました。
2.9.1	Greengrass nucleus バージョン 2.9.1 のリリース用にバージョンが更新されました。
2.9.0	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.8.1	Greengrass nucleus バージョン 2.8.1 のリリース用にバージョンが更新されました。
2.8.0	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.7.0	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.6.0	新機能 • Greengrass CLI が使用するプロセス間通信 (IPC) のオペレーションを呼び出す、カスタムコンポーネントに関するサポートが追加されました。これらの IPC オペレーションにより、ローカルでのデプロイ管理、コンポーネントの詳細の表示、およびローカルのデバッグコンソールへのサインイン用のパスワード生成を行うことができます。詳細については、「IPC: Manage local deployments and components」(IPC: ローカルのデプロイとコンポーネントを管理する) を参照してください。 バグ修正と機能向上 • 追加のマイナー修正と機能向上。

バージョン	変更
2.5.6	Greengrass nucleus バージョン 2.5.6 のリリース用にバージョンが更新されました。
2.5.5	Greengrass nucleus バージョン 2.5.5 のリリース用にバージョンが更新されました。
2.5.4	Greengrass nucleus バージョン 2.5.4 のリリース用にバージョンが更新されました。
2.5.3	Greengrass nucleus バージョン 2.5.3 のリリース用にバージョンが更新されました。
2.5.2	Greengrass nucleus バージョン 2.5.2 のリリース用にバージョンが更新されました。
2.5.1	Greengrass nucleus バージョン 2.5.1 のリリース用にバージョンが更新されました。
2.5.0	新機能 • Windows を実行するコアデバイスのサポートが追加されました。 • Windows デバイスで Greengrass CLI を使用するために、システムグループを認可するために指定できる新しい `AuthorizedWindowsGroups` 設定パラメータを追加しました。 • ローカルデプロイ用の `windowsUser` パラメータが追加されました。このパラメータを使用して、Windows コアデバイスでコンポーネントを実行するために使用するユーザーを指定できます。
2.4.0	新機能 • システムリソース制限のサポートを追加します。ローカルデプロイを作成するときに、各コンポーネントのプロセスがコアデバイスで使用できる CPU および RAM の最大使用数を設定できます。詳細については、「コンポーネントのシステムリソース制限を設定する」と「デプロイ作成コマンド」を参照してください。
2.3.0	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.2.0	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.1.0	Greengrass nucleus バージョン 2.1.0 のリリース用にバージョンが更新されました。
2.0.5	Greengrass nucleus バージョン 2.0.5 のリリース用にバージョンが更新されました。
2.0.4	Greengrass nucleus バージョン 2.0.4 のリリース用にバージョンが更新されました。
2.0.3	当初のバージョン

IP ディテクター

IP ディテクターコンポーネント (`aws.greengrass.clientdevices.IPDetector`) は以下の処理を行います。

- Greengrass コアデバイスのローカルネットワーク接続情報を監視します。この情報には、コアデバイスのネットワークエンドポイントと、MQTT ブローカーが動作するポートが含まれます。
- AWS IoT Greengrass クラウドサービス内のコアデバイスの接続情報を更新します。

クライアントデバイスは Greengrass クラウドディスカバリを使用して、関連するコアデバイスの接続情報を取得できます。その後クライアントデバイスは、正常に接続されるまで各コアデバイスへの接続を試みることができます。

> **Note**
>
> クライアントデバイスは、Greengrass コアデバイスに接続し、処理するために MQTT メッセージとデータを送信するローカル IoT デバイスです。(詳細については、ローカル IoT デバイスとやり取りする を参照してください)。

IP ディテクターコンポーネントは、コアデバイスの既存の接続情報を検出した情報に置き換えます。このコンポーネントは既存の情報を削除するため、IP ディテクターコンポーネントを使用するか、接続情報を手動で管理するか選択できます。

> **Note**
>
> IP ディテクターコンポーネントは IPv4 アドレスのみを検出します。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x
- 2.0.x

タイプ

このコンポーネントはプラグインコンポーネント (`aws.greengrass.plugin`) です。Greengrass nucleus は、nucleus と同じ Java バーチャルマシン (JVM) でこのコンポーネントを実行します。コアデバイスでこのコンポーネントのバージョンを変更するとき、nucleus が再起動します。

このコンポーネントは、Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Greengrass サービスロールを AWS アカウント に関連づけ、`iot:GetThingShadow` および `iot:UpdateThingShadow` の権限を許可する必要があります。
- コアデバイスの AWS IoT ポリシーで、`greengrass:UpdateConnectivityInfo` 権限を許可する必要があります。詳細については、「データプレーンオペレーションの AWS IoT ポリシー と クライアントデバイスをサポートするための最低限の AWS IoT ポリシー」を参照してください。
- デフォルトポートの 8883 以外のポートを使用するようにコアデバイスの MQTT ブローカーコンポーネントを設定する場合、IP ディテクター v2.1.0 以降を使用する必要があります。ブローカーが動作するポートを報告するように設定します。
- 複雑なネットワーク設定がある場合、IP ディテクターコンポーネントは、クライアントデバイスがコアデバイスに接続できるエンドポイントを識別できない場合があります。IP ディテクターコンポーネントがエンドポイントを管理できない場合、代わりにコア デバイス エンドポイントを手動で管理する必要があります。たとえば、コアデバイスが MQTT ブローカポートを転送するルーターの背後にある場合、コアデバイスのエンドポイントとしてルーターの IP アドレスを指定する必要があります。詳細については、「コアデバイスのエンドポイントを管理」を参照してください。
- IP ディテクターコンポーネントは、VPC での実行がサポートされています。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネント

のリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.8

次の表に、このコンポーネントのバージョン 2.1.8 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.13.0	ソフト

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.12.0	ソフト

2.1.6

次の表に、このコンポーネントのバージョン 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.11.0	ソフト

2.1.5

次の表に、このコンポーネントのバージョン 2.1.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.10.0	ソフト

2.1.4

次の表に、このコンポーネントのバージョン 2.1.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.9.0	ソフト

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.8.0	ソフト

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.7.0	ソフト

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.6.0	ソフト

2.1.0 and 2.0.2

次の表に、このコンポーネントのバージョン 2.1.0 および 2.0.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.5.0	ソフト

2.0.1

次の表に、このコンポーネントのバージョン 2.0.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.4.0	ソフト

2.0.0

次の表に、このコンポーネントのバージョン 2.0.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.3.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

2.1.x

`defaultPort`

(オプション) このコンポーネントが IP アドレスを検出するときに報告する MQTT ブローカーポート。デフォルトポート 8883 とは異なるポートを使用するように MQTT ブローカーを設定する場合、このパラメータを指定する必要があります。

デフォルト: 8883

`includeIPv4LoopbackAddrs`

(オプション) このオプションを有効にすると、IPv4 ループバックアドレスが検出および報告されます。これらは、`localhost` など、IP アドレスであり、デバイスが自身と通信できる場所です。このオプションは、コアデバイスとクライアントデバイスを同じシステムで実行するテスト環境で使用します。

デフォルト: `false`

`includeIPv4LinkLocalAddrs`

(オプション) このオプションを有効にすると、IPv4 のリンクローカルアドレスが検出および報告されます。このオプションは、コアデバイスのネットワークに Dynamic Host Configuration Protocol (DHCP) または静的に割り当てられた IP アドレスがない場合に使用します。

デフォルト: `false`

2.0.x

`includeIPv4LoopbackAddrs`

(オプション) このオプションを有効にすると、IPv4 ループバックアドレスが検出および報告されます。これらは、`localhost` など、IP アドレスであり、デバイスが自身と通信できる場所です。このオプションは、コアデバイスとクライアントデバイスを同じシステムで実行するテスト環境で使用します。

デフォルト: `false`

`includeIPv4LinkLocalAddrs`

(オプション) このオプションを有効にすると、IPv4 のリンクローカルアドレスが検出および報告されます。このオプションは、コアデバイスのネットワークに Dynamic Host Configuration Protocol (DHCP) または静的に割り当てられた IP アドレスがない場合に使用します。

デフォルト: `false`

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.8	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.1.7	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.6	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.1.5	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.4	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.3	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.1.2	バグ修正と機能向上 • 特定のシナリオで、このコンポーネントがログに記録するエラーメッセージを改善。 • Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.1	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.1.0	改良点 • デフォルト以外の MQTT ブローカーポートを使用できるようにする `defaultPort` パラメータを追加します。 • ログメッセージをより明確にするための更新。
2.0.2	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.0.1	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.0.0	当初のバージョン

Firehose

Firehose コンポーネント (`aws.greengrass.KinesisFirehose`) は、Amazon Data Firehose 配信ストリームを介して、Amazon S3、Amazon Redshift、Amazon OpenSearch Service などの送

信先にデータを発行します。詳細については、「Amazon Data Firehose デベロッパーガイド」の「Amazon Data Firehose とは」を参照してください。

このコンポーネントを使用して Kinesis 配信ストリームに発行するには、このコンポーネントがサブスクライブしているトピックにメッセージを発行します。デフォルトでは、このコンポーネントは `kinesisfirehose/message` および `kinesisfirehose/message/binary/#` のローカルパブリッシュ/サブスクライブトピックにサブスクライブしています。このコンポーネントをデプロイするときに、 AWS IoT Core MQTT トピックを含む他のトピックを指定できます。

> **Note**
>
> このコンポーネントは、 AWS IoT Greengrass V1 の Firehose コネクタと同様の機能を提供します。詳細については、AWS IoT Greengrass 「 V1 デベロッパーガイド」の「Firehose コネクタ」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x
- 2.0.x

タイプ

このコンポーネントは Lambda コンポーネントです (`aws.greengrass.lambda`)。Greengrass nucleus は、Lambda ランチャーコンポーネントを使用してこのコンポーネントの Lambda 関数を実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、Linux コアデバイスにのみインストールできます。

要件

このコンポーネントには次の要件があります。

- コアデバイスは、Lambda 関数を実行するための要件を満たしている必要があります。コアデバイスが、コンテナ化された Lambda 関数を実行させる場合、そのデバイスは要件を満たす必要があります。詳細については、「Lambda 関数の要件」を参照してください。
- Python バージョン 3.7 がコアデバイスにインストールされ、PATH 環境変数に追加されています。
- 次の IAM ポリシーの例で示されているように、Greengrass デバイスのロールは、`firehose:PutRecord` および `firehose:PutRecordBatch` のアクションを許可する必要があります。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "firehose:PutRecord",
        "firehose:PutRecordBatch"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:firehose:region:account-id:deliverystream/stream-name"
      ]
```

```
    }
  ]
}
```

このコンポーネントの入力メッセージペイロードのデフォルト配信ストリームを動的にオーバーライドできます。アプリケーションでこの機能を使用する場合、IAM ポリシーにはすべてのターゲットストリームをリソースとして含める必要があります。リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (たとえば、ワイルドカード * 命名スキームを使用)。

- このコンポーネントから出力データを受信するには、このコンポーネントをデプロイするときに、次の設定更新プログラムをレガシーサブスクリプションルーターのコンポーネント (aws.greengrass.LegacySubscriptionRouter) のためにマージする必要があります。この設定は、このコンポーネントがレスポンスを公開するトピックを指定します。

  Legacy subscription router v2.1.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-kinesisfirehose": {
        "id": "aws-greengrass-kinesisfirehose",
        "source": "component:aws.greengrass.KinesisFirehose",
        "subject": "kinesisfirehose/message/status",
        "target": "cloud"
      }
    }
  }
  ```

  Legacy subscription router v2.0.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-kinesisfirehose": {
        "id": "aws-greengrass-kinesisfirehose",
        "source": "arn:aws:lambda:region:aws:function:aws-greengrass-
  kinesisfirehose:version",
        "subject": "kinesisfirehose/message/status",
        "target": "cloud"
      }
    }
  }
  ```

  - *region* AWS リージョン は、使用する に置き換えます。

- *#####*を、このコンポーネントが実行する Lambda 関数のバージョンに置き換えます。Lambda 関数のバージョンを確認するには、デプロイするこのコンポーネントのバージョンの recipe を確認する必要があります。AWS IoT Greengrass コンソールで、このコンポーネントの詳細ページを開き、[Lambda function] (Lambda 関数) の key-value ペアを見つけます。このキー値のペアには、Lambda 関数の名前とバージョンが含まれます。

  > **Important**
  >
  > このコンポーネントをデプロイするたびに、レガシーサブスクリプションルーターの Lambda 関数のバージョンを更新する必要があります。これにより、デプロイするコンポーネントバージョンに正しい Lambda 関数のバージョンが使用されることが保証されます。

  詳細については、「デプロイの作成」を参照してください。

- Firehose コンポーネントは、VPC での実行がサポートされています。このコンポーネントを VPC にデプロイするには、以下が必要です。
  - Firehose コンポーネントは、VPC エンドポイントが `firehose.region.amazonaws.com`である への接続が必要です`com.amazonaws.region.kinesis-firehose`。

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
`firehose.` *`region`*`.amazonaws.com`	443	はい	Firehose にデータをアップロードします。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.6

次の表に、このコンポーネントのバージョン 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.5

次の表に、このコンポーネントのバージョン 2.1.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.4

次の表に、このコンポーネントのバージョン 2.1.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト

依存関係	互換性のあるバージョン	依存関係タイプ
トークン交換サービス	^2.0.0	ハード

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.8 - 2.1.0

次の表に、このコンポーネントのバージョン 2.0.8 および 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.7

次の表に、このコンポーネントのバージョン 2.0.7 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.6

次の表に、このコンポーネントのバージョン 2.0.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.5

次の表に、このコンポーネントのバージョン 2.0.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.4

次の表に、このコンポーネントのバージョン 2.0.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.3

次の表に、このコンポーネントのバージョン 2.0.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.3 <2.1.0	ハード
Lambda ランチャー	>=1.0.0	ハード
Lambda ランタイム	>=1.0.0	ソフト

依存関係	互換性のあるバージョン	依存関係タイプ
トークン交換サービス	>=1.0.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

> **Note**
>
> このコンポーネントのデフォルト設定には、Lambda 関数のパラメータが含まれます。デバイスにこのコンポーネントを設定するには、次のパラメータのみを編集することをお勧めします。

`lambdaParams`

このコンポーネントの Lambda 関数のパラメータを含むオブジェクト。このオブジェクトには、次の情報が含まれます。

`EnvironmentVariables`

Lambda 関数のパラメータを含むオブジェクト。このオブジェクトには、次の情報が含まれます。

`DEFAULT_DELIVERY_STREAM_ARN`

コンポーネントがデータを送信するデフォルトの Firehose 配信ストリームの ARN。入力メッセージペイロードの `delivery_stream_arn` プロパティを使用して、送信先ストリームをオーバーライドできます。

> **Note**
>
> コアデバイスロールは、すべてのターゲット配信ストリームで必要なアクションを許可する必要があります。詳細については、「要件」を参照してください。

PUBLISH_INTERVAL

(オプション) コンポーネントがバッチ処理されたデータを Firehose に発行するまでの最大待機秒数。メトリクスを受信した時点でメトリクスをパブリッシュするようにコンポーネントを設定するには (バッチ処理なし)、0 を指定します。

この値は最大 900 秒になります。

デフォルト: 10 秒

DELIVERY_STREAM_QUEUE_SIZE

(オプション) コンポーネントが同じ配信ストリームの新しいレコードを拒否するまでに、メモリに保持するレコードの最大数。

この値は 2,000 件以上に指定する必要があります。

デフォルト: 5,000 件

containerMode

(オプション) このコンポーネントのコンテナ化モード。次のオプションから選択します。

- NoContainer - コンポーネントは、分離されたランタイム環境では実行されません。
- GreengrassContainer – コンポーネントは、 AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。

デフォルト: GreengrassContainer

containerParams

(オプション) このコンポーネントのコンテナパラメータを含むオブジェクト。containerMode の GreengrassContainer を指定した場合、コンポーネントはこれらのパラメータを使用します。

このオブジェクトには、次の情報が含まれます。

memorySize

(オプション) コンポーネントに割り当てるメモリ量 (KB 単位)。

デフォルトは 64 MB (65,535 KB) です。

pubsubTopics

(オプション) コンポーネントがメッセージを受信するためにサブスクライブするトピックを含むオブジェクト。各トピックと、コンポーネントが から MQTT トピックをサブスクライブする

か、ローカルのパブリッシュ/サブスクライブトピックをサブスクライブ AWS IoT Core するかを指定できます。

このオブジェクトには、次の情報が含まれます。

`0` - これは文字列としての配列インデックスです。

次の情報が含まれるオブジェクト。

`type`

(オプション) このコンポーネントがメッセージをサブスクライブするために使用するパブリッシュ/サブスクライブメッセージングのタイプ。次のオプションから選択します。

- PUB_SUB - ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブします。このオプションを選択した場合、トピックに MQTT ワイルドカードを含めることはできません。このオプションを指定したときに、カスタムコンポーネントからメッセージを送信する方法の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。
- IOT_CORE – AWS IoT Core MQTT メッセージをサブスクライブします。このオプションを選択した場合、トピックに MQTT ワイルドカードを含めることができます。このオプションを指定したときに、カスタムコンポーネントからメッセージを送信する方法の詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。

デフォルト: PUB_SUB

`topic`

(オプション) コンポーネントがメッセージを受信するためにサブスクライブするトピック。`type` の `IotCore` を指定した場合、このトピックで MQTT ワイルドカード (+ および #) を使用できます。

Example 例: 設定マージの更新 (コンテナモード)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "DEFAULT_DELIVERY_STREAM_ARN": "arn:aws:firehose:us-
west-2:123456789012:deliverystream/mystream"
    }
```

```
  },
  "containerMode": "GreengrassContainer"
}
```

Example 例: 設定マージの更新 (コンテナモードなし)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "DEFAULT_DELIVERY_STREAM_ARN": "arn:aws:firehose:us-
west-2:123456789012:deliverystream/mystream"
    }
  },
  "containerMode": "NoContainer"
}
```

入力データ

このコンポーネントは、以下のトピックに関するストリームコンテンツを受け入れ、そのコンテンツをターゲット配信ストリームに送信します。コンポーネントは、以下の 2 種類の入力データを受け入れます。

- `kinesisfirehose/message` トピックの JSON データ。
- `kinesisfirehose/message/binary/#` トピックのバイナリデータ。

JSON データのデフォルトトピック (ローカルパブリッシュ/サブスクライブ): `kinesisfirehose/message`

メッセージは、次のプロパティを受付けます。入力メッセージは JSON 形式である必要があります。

`request`

配信ストリームに送信するデータ、およびターゲット配信ストリーム (デフォルトストリームと異なる場合)。

タイプ: 次の情報が含まれる `object`。

`data`

配信ストリームに送信するデータ。

タイプ: `string`

`delivery_stream_arn`

(オプション) ターゲット Firehose 配信ストリームの ARN。デフォルトの配信ストリームを上書きするには、このプロパティを指定します。

タイプ: `string`

`id`

リクエストの任意の ID。このプロパティを使用して、入力リクエストを出力レスポンスにマッピングします。このプロパティを指定するとき、コンポーネントはこの値に対してレスポンスオブジェクトの `id` プロパティを設定します。

タイプ: `string`

Example 入力例

```
{
  "request": {
    "delivery_stream_arn": "arn:aws:firehose:region:account-id:deliverystream/
stream2-name",
    "data": "Data to send to the delivery stream."
  },
  "id": "request123"
}
```

バイナリデータのデフォルトトピック (ローカルパブリッシュ/サブスクライブ): `kinesisfirehose/message/binary/#`

このトピックを使用して、バイナリデータを含むメッセージを送信します。コンポーネントはバイナリデータを解析しません。コンポーネントはデータをそのままストリーミングします。

入力リクエストを出力レスポンスにマッピングするには、メッセージトピックの `#` ワイルドカードを任意のリクエスト ID に置き換えます。例えば、メッセージを `kinesisfirehose/message/binary/request123` に発行する場合、レスポンスオブジェクトの `id` プロパティを `request123` に設定します。

リクエストをレスポンスにマッピングしない場合は、メッセージを `kinesisfirehose/message/binary/` に発行できます。末尾にスラッシュ (`/`) を含めてください。

出力データ

このコンポーネントは、デフォルトで次の MQTT トピックに出力データとしてレスポンスを公開します。このトピックは、[legacy subscription router component] (レガシーサブスクリプションルーターコンポーネント) の設定で `subject` として指定する必要があります。カスタムコンポーネントでこのトピックに関するメッセージへサブスクライブする方法の詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。

デフォルトトピック (AWS IoT Core MQTT): `kinesisfirehose/message/status`

Example 出力例

応答には、バッチで送信される各データレコードのステータスが含まれます。

```
{
  "response": [
    {
      "ErrorCode": "error",
      "ErrorMessage": "test error",
      "id": "request123",
      "status": "fail"
    },
    {
      "firehose_record_id": "xyz2",
      "id": "request456",
      "status": "success"
    },
    {
      "firehose_record_id": "xyz3",
      "id": "request890",
      "status": "success"
    }
  ]
}
```

> **Note**
>
> コンポーネントが、接続エラーなどの再試行可能なエラーを検出した場合、次のバッチでパブリッシュを再試行します。

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

```
/greengrass/v2/logs/aws.greengrass.KinesisFirehose.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。を AWS IoT Greengrass ルートフォルダへのパス*/greengrass/v2*に置き換えます。

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.KinesisFirehose.log
  ```

ライセンス

このコンポーネントには、次のサードパーティーソフトウェア/ライセンス品が含まれています。

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF ライセンス
- docutils/BSD ライセンス、GNU 一般パブリックライセンス (GPL)、Python Software Foundation ライセンス、パブリックドメイン
- jmespath/MIT ライセンス
- s3transfer/Apache License 2.0
- urllib3/MIT ライセンス

このコンポーネントは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされています。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.7	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.1.6	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.5	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.4	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.3	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.1.1	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.0	新機能 • HTTPS ネットワークプロキシ設定へのサポートが追加されました。詳細については、ポート 443 での接続またはネットワークプロキシを通じた接続およびコアデバイスが HTTPS プロキシを信頼できるようにするを参照してください。
2.0.8	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.0.7	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.0.6	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.0.5	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.0.4	Greengrass nucleus バージョン 2.1.0 のリリース用にバージョンが更新されました。
2.0.3	当初のバージョン

以下も参照してください。

- 「Amazon Data Firehose デベロッパーガイド」の「Amazon Data Firehose とは」

Lambda ランチャー

Lambda ランチャーコンポーネント (`aws.greengrass.LambdaLauncher`) は、 AWS IoT Greengrass コアデバイスの関数を開始および停止します AWS Lambda 。また、このコンポーネントはあらゆるコンテナ化を設定し、指定されたユーザーとしてプロセスを実行します。

> **Note**
>
> Lambda 関数コンポーネントをコアデバイスにデプロイすると、デプロイにはこのコンポーネントも含められます。詳細については、「AWS Lambda 関数を実行する」を参照してください。

トピック

- バージョン
- タイプ
- オペレーティングシステム
- 要件
- 依存関係
- 構成
- ローカルログファイル

- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、Linux コアデバイスにのみインストールできます。

要件

このコンポーネントには次の要件があります。

- コアデバイスは、Lambda 関数を実行するための要件を満たしている必要があります。コアデバイスが、コンテナ化された Lambda 関数を実行させる場合、そのデバイスは要件を満たす必要があります。詳細については、「Lambda 関数の要件」を参照してください。
- Lambda ランチャーコンポーネントは、VPC での実行がサポートされています。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.0.11 – 2.0.13

次の表に、このコンポーネントのバージョン 2.0.11 から 2.0.13 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Lambda マネージャー	>=2.0.0 <2.4.0	ハード

2.0.9 – 2.0.10

次の表に、このコンポーネントのバージョン 2.0.9 から 2.0.10 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Lambda マネージャー	>=2.0.0 <2.3.0	ハード

2.0.4 - 2.0.8

次の表に、このコンポーネントのバージョン 2.0.4 から 2.0.8 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Lambda マネージャー	>=2.0.0 <2.2.0	ハード

2.0.3

次の表に、このコンポーネントのバージョン 2.0.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Lambda マネージャー	>=2.0.3 <2.1.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントに設定パラメータはありません。

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

```
/greengrass/v2/logs/lambdaFunctionComponentName.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。を AWS IoT Greengrass ルートフォルダへのパス*/greengrass/v2*に置き換え、*lambdaFunctionComponentName* をこのコンポーネントが起動する Lambda 関数コンポーネントの名前に置き換えます。

  ```
  sudo tail -f /greengrass/v2/logs/lambdaFunctionComponentName.log
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.0.13	バグ修正と機能向上 一般的なバグ修正と機能強化。
2.0.12	バグ修正と機能向上 前のプロセスが正しく停止しなかった場合に、Lambda ランチャーがエラーをスローすることがある問題を修正しました。
2.0.11	Lambda マネージャー 2.3.0 のサポート。
2.0.10	バグ修正と機能向上 • 一般的なバグ修正と機能強化。

バージョン	変更
2.0.9	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.0.8	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.0.7	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.0.6	一般的なパフォーマンス向上とバグ修正。
2.0.4	バグ修正と機能向上 • コンポーネントが `AddGroupOwner` Lambda 関数コンテナを正しく渡さない問題を修正しました。
2.0.3	当初のバージョン

Lambda マネージャー

Lambda マネージャーコンポーネント (`aws.greengrass.LambdaManager`) は、Greengrass コアデバイスで実行される AWS Lambda 関数の作業項目とプロセス間通信を管理します。

> **Note**
>
> Lambda 関数コンポーネントをコアデバイスにデプロイすると、デプロイにはこのコンポーネントも含められます。詳細については、「AWS Lambda 関数を実行する」を参照してください。

トピック

- 構成
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

オペレーティングシステム

このコンポーネントは、Linux コアデバイスにのみインストールできます。

タイプ

このコンポーネントはプラグインコンポーネント (`aws.greengrass.plugin`) です。Greengrass nucleus は、nucleus と同じ Java バーチャルマシン (JVM) でこのコンポーネントを実行します。コアデバイスでこのコンポーネントのバージョンを変更するとき、nucleus が再起動します。

このコンポーネントは、Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

詳細については、「コンポーネントタイプ」を参照してください。

要件

このコンポーネントには次の要件があります。

- コアデバイスは、Lambda 関数を実行するための要件を満たしている必要があります。コアデバイスが、コンテナ化された Lambda 関数を実行させる場合、そのデバイスは要件を満たす必要があります。詳細については、「Lambda 関数の要件」を参照してください。
- Lambda マネージャーコンポーネントは、VPC での実行がサポートされています。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.3.2

次の表に、このコンポーネントのバージョン 2.3.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト

2.2.10 and 2.3.1

次の表に、このコンポーネントのバージョン 2.2.10 および 2.3.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト

2.2.8 and 2.2.9

次の表に、このコンポーネントのバージョン 2.2.8 および 2.2.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト

2.2.7

次の表に、このコンポーネントのバージョン 2.2.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト

2.2.6

次の表に、このコンポーネントのバージョン 2.2.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト

2.2.5

次の表に、このコンポーネントのバージョン 2.2.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト

2.2.4

次の表に、このコンポーネントのバージョン 2.2.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト

2.2.1 - 2.2.3

次の表に、このコンポーネントのバージョン 2.2.1 から 2.2.3 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト

2.2.0

次の表に、このコンポーネントのバージョン 2.2.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.6.0	ソフト

2.1.3 and 2.1.4

次の表に、このコンポーネントのバージョン 2.1.3 および 2.1.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト

2.1.0

次の表に、このコンポーネントのバージョン 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト

2.0.x

次の表に、このコンポーネントのバージョン 2.0.x の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.3 <2.1.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

`logHandlerMode`

> **Note**
>
> Lambda マネージャーバージョン 2.3.0 以降のみ

使用する Lambda ログマネージャーの実装を選択するために使用されます。Lambda ログを読み取るために使用するスレッドの数を減らすには、`optimized` に値を設定します。

`getResultTimeoutInSecond`

(オプション) Lambda 関数がタイムアウトするまでの最大実行時間 (秒)。

デフォルト: `60`

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

```
/greengrass/v2/logs/greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。をAWS IoT Greengrassルートフォルダへのパス*/greengrass/v2*に置き換えます。

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.3.2	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.3.1	バグ修正と機能向上 • 特定のエラーのログレベルを調整します。
2.3.0	新機能 • ログハンドラは CPU 負荷を軽減するように最適化されました。この機能を使用するには、新しい設定オプション `logHandlerMode` に `optimized` を設定します。 バグ修正と機能向上 • `WorkQueueFullException` の完全なスタックトレースをログに記録しなくなったため、ログとパフォーマンスが向上しました。 • シャットダウンタイムアウトを防ぐために、Lambda シャットダウンタイムアウトを 15 秒から 300 秒に設定します。 • オンデマンド Lambda が設定を変更した後に再起動できない問題を修正します。

バージョン	変更
2.2.11	バグ修正と機能向上 • Lambda LegacySubscriptionRouter 設定が変更されても設定が更新されない問題を修正しました。
2.2.10	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.2.9	バグ修正と機能向上 クロックが歪んでいるためにポート番号が壊れている問題を修正します。
2.2.8	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.2.7	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.2.6	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.2.5	新機能 • ローカルの公開/サブスクライブメッセージにサブスクライブするイベントソースで、MQTT トピックのワイルドカードが使用できるようになりました。 この機能を使用するには、Greengrass nucleus コンポーネントの v2.6.0 以降が必要です。 • Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.2.4	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.2.3	バグ修正と機能向上 • Lambda 関数の複数のインスタンスが 1 つの cgroup を共有する問題を修正しました。このコンポーネントは、cgroups を使用して Lambda 関数のリソース使用量を管理します。
2.2.2	バグ修正と機能向上 • 特定のシナリオで、固定された Lambda 関数コンポーネントが予期せず再起動する問題を修正しました。
2.2.1	バグ修正と機能向上 • 依存関係解決の問題を修正するため、このコンポーネントの Greengrass nucleus 依存バージョンの制約を変更します。
2.2.0	バグ修正と機能向上 • 再起動後に Lambda 関数がログを書き込めない問題を修正しました。 • トピックにワイルドカードがある場合、レガシーサブスクリプションルータが重複するメッセージを送信する問題を修正しました。 • ピン留めされていない Lambda 関数が AWS IoT Device SDK で Greengrass プロセス間通信 (IPC) ライブラリを使用できない問題を修正しました。
2.1.4	バグ修正と機能向上 • NodeJS ランタイムを使用する Lambda 関数が 1 つのメッセージのみを処理することの原因となる問題を修正しました。 • Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.1.3	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.1.1	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.1.0	Greengrass nucleus バージョン 2.1.0 のリリース用にバージョンが更新されました。
2.0.3	当初のバージョン

Lambda ランタイム

Lambda ランタイムコンポーネント (`aws.greengrass.LambdaRuntimes`) は、Greengrass コアデバイスが AWS Lambda 関数を実行するために使用するランタイムを提供します。

> **Note**
>
> Lambda 関数コンポーネントをコアデバイスにデプロイすると、デプロイにはこのコンポーネントも含められます。詳細については、「AWS Lambda 関数を実行する」を参照してください。

トピック

- バージョン
- タイプ
- オペレーティングシステム
- 要件
- 依存関係
- 構成
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、Linux コアデバイスにのみインストールできます。

要件

このコンポーネントには次の要件があります。

- コアデバイスは、Lambda 関数を実行するための要件を満たしている必要があります。コアデバイスが、コンテナ化された Lambda 関数を実行させる場合、そのデバイスは要件を満たす必要があります。詳細については、「Lambda 関数の要件」を参照してください。
- Lambda ランタイムコンポーネントは、VPC での実行がサポートされています。

依存関係

このコンポーネントに依存関係はありません。

構成

このコンポーネントに設定パラメータはありません。

ローカルログファイル

このコンポーネントはログを出力しません。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.0.8	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.0.7	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.0.6	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.0.5	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.0.4	Greengrass nucleus バージョン 2.1.0 のリリース用にバージョンが更新されました。
2.0.3	当初のバージョン

レガシーサブスクリプションルーター

レガシーサブスクリプションルータ (`aws.greengrass.LegacySubscriptionRouter`) は、Greengrass コアデバイスのサブスクリプションを管理します。サブスクリプションは AWS IoT Greengrass V1 の機能で、Lambda 関数がコアデバイスの MQTT メッセージングに使用できるトピックを定義するものです。詳細については、「AWS IoT Greengrass V1 デベロッパーガイド」の「MQTT メッセージングワークフローにおけるマネージドサブスクリプション」を参照してください。

このコンポーネントを使用して、AWS IoT Greengrass Core SDK を使用するコネクタコンポーネントと Lambda 関数コンポーネントのサブスクリプションを有効にできます。

> **Note**
>
> レガシーサブスクリプションルーターコンポーネントは、Lambda 関数が AWS IoT Greengrass Core SDK の `publish()` 関数を使用する場合にのみ必要です。AWS IoT Device SDK V2 でプロセス間通信 (IPC) インターフェイスを使用するように Lambda 関数コードを更新する場合、レガシーサブスクリプションルーターコンポーネントをデプロイする必要はありません。詳細については、次のプロセス間通信サービスを参照してください。
>
> - ローカルメッセージをパブリッシュ/サブスクライブする

- AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x
- 2.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、Linux コアデバイスにのみインストールできます。

要件

このコンポーネントには次の要件があります。

- レガシーサブスクリプションルーターは、VPC での実行がサポートされています。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.11

次の表に、このコンポーネントのバージョン 2.1.11 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト

2.1.10

次の表に、このコンポーネントのバージョン 2.1.10 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト

2.1.9

次の表に、このコンポーネントのバージョン 2.1.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト

2.1.8

次の表に、このコンポーネントのバージョン 2.1.8 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト

2.1.6

次の表に、このコンポーネントのバージョン 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト

2.1.5

次の表に、このコンポーネントのバージョン 2.1.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト

2.1.4

次の表に、このコンポーネントのバージョン 2.1.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト

2.1.0

次の表に、このコンポーネントのバージョン 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト

2.0.3

次の表に、このコンポーネントのバージョン 2.0.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.3 <2.1.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

v2.1.x

`subscriptions`

(オプション) コアデバイスで有効にするサブスクリプション。これはオブジェクトで、各キーは一意の ID であり、各値はそのコネクタのサブスクリプションを定義するオブジェクトです。V1 コネクタコンポーネントまたは AWS IoT Greengrass Core SDK を使用する Lambda 関数をデプロイするときは、サブスクリプションを設定する必要があります。

各サブスクリプションオブジェクトには、次の情報が含まれます:

`id`

このサブスクリプションの一意の ID。この ID は、このサブスクリプションオブジェクトのキーと一致する必要があります。

`source`

AWS IoT Greengrass Core SDKを 使用して、`subject` で指定したトピックに関する MQTT メッセージの発行を行う Lambda 関数です。次のいずれかを指定します。

- コアデバイスでの Lambda 関数コンポーネントの名前。**`component:com.example.HelloWorldLambda`** などの `component:` プレフィックスを付けてコンポーネント名を指定します。
- コアデバイスの Lambda 関数の Amazon リソースネーム (ARN)。

> **Important**
>
> Lambda 関数のバージョンが変更された場合は、関数の新しいバージョンでサブスクリプションを設定する必要があります。これを怠ると、バージョンがサブスクリプションと一致しないうちは、このコンポーネントはメッセージをルーティングしません。
> インポートする関数のバージョンが含まれた Amazon リソースネーム (ARN) を指定する必要があります。$LATEST のようなバージョンエイリアスは使用できません。

V1 コネクタコンポーネントのサブスクリプションをデプロイするには、コンポーネントの名前またはコネクタコンポーネントの Lambda 関数の ARN を指定します。

`subject`

ソースとターゲットがメッセージを発行および受信できる MQTT トピックまたはトピックフィルター。この値は + および # トピックのワイルドカードをサポートしています。

`target`

`subject` で指定したトピックに関する MQTT メッセージを受信するターゲット。サブスクリプションは、`source` 関数が AWS IoT Core またはコアデバイスの Lambda 関数に MQTT メッセージを発行することを指定します。次のいずれかを指定します。

- `cloud`。`source` 関数は MQTT メッセージを AWS IoT Core に発行します。
- コアデバイスでの Lambda 関数コンポーネントの名前。**`component:com.example.HelloWorldLambda`** などの `component:` プレフィックスを付けてコンポーネント名を指定します。
- コアデバイスの Lambda 関数の Amazon リソースネーム (ARN)。

> **Important**
>
> Lambda 関数のバージョンが変更された場合は、関数の新しいバージョンでサブスクリプションを設定する必要があります。これを怠ると、バージョンがサブスクリプションと一致しないうちは、このコンポーネントはメッセージをルーティングしません。

インポートする関数のバージョンが含まれた Amazon リソースネーム (ARN) を指定する必要があります。$LATEST のようなバージョンエイリアスは使用できません。

デフォルト: サブスクリプションなし

Example 設定更新の例 (AWS IoT Core に対するサブスクリプションの定義)

以下の例は、com.example.HelloWorldLambda Lambda 関数コンポーネントが AWS IoT Core に hello/world トピックに関する MQTT メッセージを公開することを指定します。

```
{
  "subscriptions": {
    "Greengrass_HelloWorld_to_cloud": {
      "id": "Greengrass_HelloWorld_to_cloud",
      "source": "component:com.example.HelloWorldLambda",
      "subject": "hello/world",
      "target": "cloud"
    }
  }
}
```

Example 設定更新の例 (別の Lambda 関数に対するサブスクリプションの定義)

以下の例は、com.example.HelloWorldLambda Lambda 関数コンポーネントが com.example.MessageRelay Lambda 関数コンポーネントに hello/world トピックに関する MQTT メッセージを公開することを指定します。

```
{
  "subscriptions": {
    "Greengrass_HelloWorld_to_MessageRelay": {
      "id": "Greengrass_HelloWorld_to_MessageRelay",
      "source": "component:com.example.HelloWorldLambda",
      "subject": "hello/world",
      "target": "component:com.example.MessageRelay"
    }
  }
}
```

v2.0.x

`subscriptions`

(オプション) コアデバイスで有効にするサブスクリプション。これはオブジェクトで、各キーは一意の ID であり、各値はそのコネクタのサブスクリプションを定義するオブジェクトです。V1 コネクタコンポーネントまたは AWS IoT Greengrass Core SDK を使用する Lambda 関数をデプロイするときは、サブスクリプションを設定する必要があります。

各サブスクリプションオブジェクトには、次の情報が含まれます:

`id`

このサブスクリプションの一意の ID。この ID は、このサブスクリプションオブジェクトのキーと一致する必要があります。

`source`

AWS IoT Greengrass Core SDKを 使用して、`subject` で指定したトピックに関する MQTT メッセージの発行を行う Lambda 関数です。次を指定します:

- コアデバイスの Lambda 関数の Amazon リソースネーム (ARN)。

> **Important**
>
> Lambda 関数のバージョンが変更された場合は、関数の新しいバージョンでサブスクリプションを設定する必要があります。これを怠ると、バージョンがサブスクリプションと一致しないうちは、このコンポーネントはメッセージをルーティングしません。
> インポートする関数のバージョンが含まれた Amazon リソースネーム (ARN) を指定する必要があります。`$LATEST` のようなバージョンエイリアスは使用できません。

V1 コネクタコンポーネントのサブスクリプションをデプロイするには、コネクタコンポーネントの Lambda 関数の ARN を指定します。

`subject`

ソースとターゲットがメッセージを発行および受信できる MQTT トピックまたはトピックフィルター。この値は + および # トピックのワイルドカードをサポートしています。

`target`

`subject` で指定したトピックに関する MQTT メッセージを受信するターゲット。サブスクリプションは、`source` 関数が AWS IoT Core またはコアデバイスの Lambda 関数に MQTT メッセージを発行することを指定します。次のいずれかを指定します。

- `cloud`。`source` 関数は MQTT メッセージを AWS IoT Core に発行します。
- コアデバイスの Lambda 関数の Amazon リソースネーム (ARN)。

> **Important**
>
> Lambda 関数のバージョンが変更された場合は、関数の新しいバージョンでサブスクリプションを設定する必要があります。これを怠ると、バージョンがサブスクリプションと一致しないうちは、このコンポーネントはメッセージをルーティングしません。
> インポートする関数のバージョンが含まれた Amazon リソースネーム (ARN) を指定する必要があります。`$LATEST` のようなバージョンエイリアスは使用できません。

デフォルト: サブスクリプションなし

Example 設定更新の例 (AWS IoT Core に対するサブスクリプションの定義)

以下の例は、`Greengrass_HelloWorld` 関数が AWS IoT Core に `hello/world` トピックに関する MQTT メッセージを公開することを指定します。

```
"subscriptions": {
  "Greengrass_HelloWorld_to_cloud": {
    "id": "Greengrass_HelloWorld_to_cloud",
    "source": "arn:aws:lambda:us-
west-2:123456789012:function:Greengrass_HelloWorld:5",
    "subject": "hello/world",
    "target": "cloud"
  }
}
```

Example 設定更新の例 (別の Lambda 関数に対するサブスクリプションの定義)

以下の例は、`Greengrass_HelloWorld` 関数が `Greengrass_MessageRelay` に `hello/world` トピックに関する MQTT メッセージを公開することを指定します。

```
"subscriptions": {
  "Greengrass_HelloWorld_to_MessageRelay": {
    "id": "Greengrass_HelloWorld_to_MessageRelay",
    "source": "arn:aws:lambda:us-
west-2:123456789012:function:Greengrass_HelloWorld:5",
    "subject": "hello/world",
    "target": "arn:aws:lambda:us-
west-2:123456789012:function:Greengrass_MessageRelay:5"
  }
}
```

ローカルログファイル

このコンポーネントはログを出力しません。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.11	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.1.10	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.9	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.8	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.7	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.6	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.1.5	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.4	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.1.3	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.1.1	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.1.0	バグ修正と機能向上 • `source` と `target` の ARN の代わりにコンポーネント名を指定するサポートを追加しました。サブスクリプションのコンポーネント名を指定する場合、Lambda 関数のバージョンが変更されるたびにサブスクリプションを再設定する必要はありません。
2.0.3	当初のバージョン

ローカルデバッグコンソール

ローカルデバッグコンソールコンポーネント (`aws.greengrass.LocalDebugConsole`) は、AWS IoT Greengrass コアデバイスとそのコンポーネントに関する情報を表示するローカルダッシュボードを提供します。このダッシュボードを使用し、コアデバイスをデバッグしてローカルコンポーネントを管理できます。

> **Important**
>
> このコンポーネントは、本番環境ではなく、開発環境でのみで使用することをお勧めします。このコンポーネントは、通常、本番環境では必要とされない情報や操作へのアクセスを

提供します。このコンポーネントを必要なコアデバイスにのみデプロイして、最小特権の原則に従います。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

タイプ

このコンポーネントはプラグインコンポーネント (`aws.greengrass.plugin`) です。Greengrass nucleus は、nucleus と同じ Java バーチャルマシン (JVM) でこのコンポーネントを実行します。コアデバイスでこのコンポーネントのバージョンを変更するとき、nucleus が再起動します。

このコンポーネントは、Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- ユーザー名とパスワードを使用してダッシュボードにサインインします。debug であるユーザー名は、お客様に提供されます。AWS IoT Greengrass CLI を使用して、コアデバイスのダッシュボードでお客様を認証する一時パスワードを作成する必要があります。AWS IoT Greengrass CLI を使用して、ローカルデバッグコンソールを使用できるようにする必要があります。詳細については、「Greengrass CLI の要件」を参照してください。パスワードを生成してサインインする方法の詳細については、「ローカルデバッグコンソールコンポーネントの使用方法」を参照してください。
- ローカルデバッグコンソールコンポーネントは、VPC での実行がサポートされています。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.4.1

次の表に、このコンポーネントのバージョン 2.4.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.10.0 <2.13.0	ハード

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass CLI	>=2.10.0 <2.13.0	ハード

2.4.0

次の表に、このコンポーネントのバージョン 2.4.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.10.0 <2.12.0	ハード
Greengrass CLI	>=2.10.0 <2.12.0	ハード

2.3.0 and 2.3.1

次の表に、このコンポーネントのバージョン 2.3.0 および 2.3.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.10.0 <2.12.0	ハード
Greengrass CLI	>=2.10.0 <2.12.0	ハード

2.2.9

次の表に、このコンポーネントのバージョン 2.2.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.12.0	ハード
Greengrass CLI	>=2.1.0 <2.12.0	ハード

2.2.8

次の表に、このコンポーネントのバージョン 2.2.8 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.11.0	ハード
Greengrass CLI	>=2.1.0 <2.11.0	ハード

2.2.7

次の表に、このコンポーネントのバージョン 2.2.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.10.0	ハード
Greengrass CLI	>=2.1.0 <2.10.0	ハード

2.2.6

次の表に、このコンポーネントのバージョン 2.2.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.9.0	ハード
Greengrass CLI	>=2.1.0 <2.9.0	ハード

2.2.5

次の表に、このコンポーネントのバージョン 2.2.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.8.0	ハード
Greengrass CLI	>=2.1.0 <2.8.0	ハード

2.2.4

次の表に、このコンポーネントのバージョン 2.2.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.7.0	ハード
Greengrass CLI	>=2.1.0 <2.7.0	ハード

2.2.3

次の表に、このコンポーネントのバージョン 2.2.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.6.0	ハード
Greengrass CLI	>=2.1.0 <2.6.0	ハード

2.2.2

次の表に、このコンポーネントのバージョン 2.2.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.5.0	ハード
Greengrass CLI	>=2.1.0 <2.5.0	ハード

2.2.1

次の表に、このコンポーネントのバージョン 2.2.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.4.0	ハード

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass CLI	>=2.1.0 <2.4.0	ハード

2.2.0

次の表に、このコンポーネントのバージョン 2.2.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.3.0	ハード
Greengrass CLI	>=2.1.0 <2.3.0	ハード

2.1.0

次の表に、このコンポーネントのバージョン 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.2.0	ハード
Greengrass CLI	>=2.1.0 <2.2.0	ハード

2.0.x

次の表に、このコンポーネントのバージョン 2.0.x の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.3 <2.1.0	ソフト
Greengrass CLI	>=2.0.3 <2.1.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

v2.1.x - v2.4.x

`httpsEnabled`

(オプション) ローカルデバッグコンソールに HTTPS 通信を有効にできます。HTTPS 通信を有効にした場合、ローカルデバッグコンソールは自己署名証明書を作成します。Web ブラウザは、自己署名証明書を使用する ウェブサイトのセキュリティ警告が表示するため、証明書を手動で認証する必要があります。次に、警告をバイパスできます。詳細については、「使用方法」を参照してください。

デフォルト: `true`

`port`

(任意) ローカルデバッグコンソールを提供するポート。

デフォルト: `1441`

`websocketPort`

(オプション) ローカルデバッグコンソールに使用するウェブソケットポート。

デフォルト: `1442`

`bindHostname`

(オプション) ローカルデバッグコンソールに使用するホスト名。

Docker コンテナで AWS IoT Greengrass Core ソフトウェアを実行した場合、Docker コンテナの外でローカルデバッグコンソールを開くことができるように、このパラメータを `0.0.0.0` に設定します。

デフォルト: `localhost`

Example 例: 設定マージの更新

次の設定の例では、デフォルト以外のポートでローカルデバッグコンソールを開いて、HTTPS を無効にするように指定しています。

```
{
  "httpsEnabled": false,
  "port": "10441",
  "websocketPort": "10442"
}
```

v2.0.x

`port`

(任意) ローカルデバッグコンソールを提供するポート。

デフォルト: `1441`

`websocketPort`

(オプション) ローカルデバッグコンソールに使用するウェブソケットポート。

デフォルト: `1442`

`bindHostname`

(オプション) ローカルデバッグコンソールに使用するホスト名。

Docker コンテナで AWS IoT Greengrass Core ソフトウェアを実行した場合、Docker コンテナの外でローカルデバッグコンソールを開くことができるように、このパラメータを `0.0.0.0` に設定します。

デフォルト: `localhost`

Example 例: 設定マージの更新

次の設定の例では、デフォルト以外のポートでローカルデバッグコンソールを開くように指定しています。

```
{
  "port": "10441",
  "websocketPort": "10442"
}
```

使用方法

ローカルデバッグコンソールを使用するには、Greengrass CLI からセッションを作成します。セッションを作成するとき、Greengrass CLI はローカルデバッグコンソールへのサインインに使用できるユーザー名と一時パスワードを提供します。

次の指示に従って、コアデバイスまたは開発コンピュータでローカルデバッグコンソールを開きます。

v2.1.x - v2.4.x

バージョン 2.1.0 以降では、ローカルデバッグコンソールはデフォルトで HTTPS を使用します。HTTPS を有効にするとき、ローカルデバッグコンソールは自己署名証明書を作成して接続を保護します。この自己署名証明書のため、ローカルデバッグコンソールを開くと、ウェブブラウザにセキュリティ警告が表示されます。Greengrass CLI を使用してセッションを作成するとき、出力に証明書のフィンガープリントが含まれるため、証明書が正当であり、接続が安全であることを確認できます。

HTTPS を無効にできます。詳細については、「ローカルデバッグコンソールの設定」を参照してください。

ローカルデバッグコンソールを開くには

1. (オプション) 開発コンピュータでローカルデバッグコンソールを確認するには、コンソールのポートを SSH 経由で転送できます。ただし、最初にコアデバイスの SSH 設定ファイルに `AllowTcpForwarding` オプションを有効にする必要があります。デフォルトでは、このオプションは有効になっています。開発用コンピュータに次のコマンドを実行して、開発用コンピュータの `localhost:1441` でダッシュボードを確認します。

   ```
   ssh -L 1441:localhost:1441 -L 1442:localhost:1442 username@core-device-ip-address
   ```

   > **Note**
   >
   > デフォルトのポートを 1441 と 1442 から変更できます。詳細については、「ローカルデバッグコンソールの設定」を参照してください。

2. セッションを作成してローカルデバッグコンソールを使用します。セッションを作成するとき、認証に使用するパスワードを生成します。このコンポーネントを使用して重要な情報

を確認して、コアデバイスで操作を実行するため、ローカルデバッグコンソールはセキュリティを強化するためにパスワードが必要です。コンポーネント設定で HTTPS を有効にすると、ローカルデバッグ コンソールも接続を保護するための証明書を作成します。HTTPS はデフォルトで有効になっています。

AWS IoT Greengrass CLI を使用してセッションを作成します。このコマンドは、8 時間後に有効期限が切れるランダムな 43 文字のパスワードを生成します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrass V2ルートフォルダへのパスに置き換えます。

Linux or Unix

```
sudo /greengrass/v2/bin/greengrass-cli get-debug-password
```

Windows

```
C:\greengrass\v2\bin\greengrass-cli get-debug-password
```

ローカルデバッグコンソールが HTTPS を使用するように設定した場合、コマンド出力は次の例のようになります。ローカルデバッグコンソールを開くとき、証明書フィンガープリントを使用して接続が安全であることを確認します。

```
Username: debug
Password: bEDp3MOHdj8ou2w5de_sCBI2XAaguy3a8XxREXAMPLE
Password expires at: 2021-04-01T17:01:43.921999931-07:00
The local debug console is configured to use TLS security. The certificate is
 self-signed so you will need to bypass your web browser's security warnings to
 open the console.
Before you bypass the security warning, verify that the certificate fingerprint
 matches the following fingerprints.
SHA-256: 15 0B 2C E2 54 8B 22 DE 08 46 54 8A B1 2B 25 DE FB 02 7D 01 4E 4A 56 67
 96 DA A6 CC B1 D2 C4 1B
SHA-1: BC 3E 16 04 D3 80 70 DA E0 47 25 F9 90 FA D6 02 80 3E B5 C1
```

デバッグビューコンポーネントは 8 時間続くセッションを作成します。その後、ローカルデバッグコンソールを再度再度するには、新しいパスワードを生成する必要があります。

3. ダッシュボードを開いてサインインします。Greengrass コアデバイスでダッシュボードを確認します。または、 SSH 経由でポートを転送する場合、開発コンピュータで確認します。次のいずれかを行います:

- デフォルト設定であるローカルデバッグコンソールで HTTPS を有効にした場合、次の手順を実行します。

  a. コアデバイスで `https://localhost:1441` を開きます。または、 SSH 経由でポートを転送した場合、開発コンピュータで開きます。

     ブラウザに、無効なセキュリティ証明書に関するセキュリティ警告が表示されることがあります。

  b. ブラウザにセキュリティ警告が表示された場合、証明書が正当であることを確認してセキュリティ警告をバイパスします。以下の操作を実行します。

     i. 証明書の SHA-256 または SHA-1 フィンガープリントを検索し、`get-debug-password` コマンドは以前に出力した SHA-256 または SHA-1 フィンガープリントと一致することを確認します。ブラウザは、一方または両方のフィンガープリントを提供することがあります。証明書とそのフィンガープリントを確認するには、ブラウザのマニュアルを参照してください。一部のブラウザでは、証明書のフィンガープリントはサムプリントと呼ばれます。

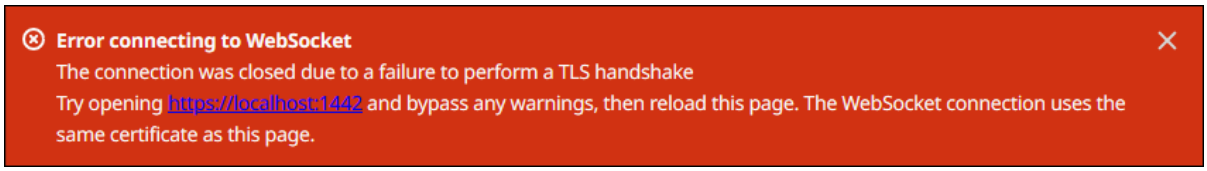
     > **Note**
     >
     > 証明書のフィンガープリントが一致しない場合、Step 2 に移動して新しいセッションを作成します。証明書のフィンガープリントがそれでも一致しない場合、接続が安全ではない可能性があります。

     ii. 証明書のフィンガープリントが一致する場合、ブラウザのセキュリティ警告をバイパスして、ローカルデバッグコンソールを開きます。ブラウザのセキュリティ警告をバイパスするには、ブラウザのマニュアルを参照してください。

  c. `get-debug-password` コマンドが以前に出力したユーザー名とパスワードを使用して、ウェブサイトにサインインします。

     ローカルデバッグコンソールが開きます。

  d. TLS ハンドシェイクが失敗した WebSocket ためにローカルデバッグコンソールに接続できないというエラーが表示された場合は、 WebSocket URL の自己署名セキュリティ警告をバイパスする必要があります。



以下の操作を実行します。

i. ローカルデバッグコンソールを開いたときと同じブラウザで `https://localhost:1442` を開きます。

ii. 証明書を確認してセキュリティ警告をバイパスします。

警告をバイパスすると、ブラウザに HTTP 404 ページが表示されることがあります。

iii. 再度 `https://localhost:1441` を開きます。

ローカルデバッグコンソールには、コアデバイスに関する情報が表示されます。

- ローカルデバッグコンソールで HTTPS を無効にした場合、次の手順を実行します。

a. コアデバイスで `http://localhost:1441` を開きます。または、SSH 経由でポートを転送した場合、開発コンピュータで開きます。

b. `get-debug-password` コマンドが以前に出力したユーザー名とパスワードを使用して、ウェブサイトにサインインします。

ローカルデバッグコンソールが開きます。

v2.0.x

ローカルデバッグコンソールを開くには

1. (オプション) 開発コンピュータでローカルデバッグコンソールを確認するには、コンソールのポートを SSH 経由で転送できます。ただし、最初にコアデバイスの SSH 設定ファイルに `AllowTcpForwarding` オプションを有効にする必要があります。デフォルトでは、このオプションは有効になっています。開発用コンピュータに次のコマンドを実行して、開発用コンピュータの `localhost:1441` でダッシュボードを確認します。

```
ssh -L 1441:localhost:1441 -L 1442:localhost:1442 username@core-device-ip-address
```

> **Note**
>
> デフォルトのポートを 1441 と 1442 から変更できます。詳細については、「ローカルデバッグコンソールの設定」を参照してください。

2. セッションを作成してローカルデバッグコンソールを使用します。セッションを作成するとき、認証に使用するパスワードを生成します。このコンポーネントを使用して重要な情報を確認して、コアデバイスで操作を実行するため、ローカルデバッグコンソールはセキュリティを強化するためにパスワードが必要です。

   AWS IoT Greengrass CLI を使用してセッションを作成します。このコマンドは、8 時間後に有効期限が切れるランダムな 43 文字のパスワードを生成します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrass V2ルートフォルダへのパスに置き換えます。

   Linux or Unix

   ```
   sudo /greengrass/v2/bin/greengrass-cli get-debug-password
   ```

   Windows

   ```
   C:\greengrass\v2\bin\greengrass-cli get-debug-password
   ```

   コマンド出力は、次の例のようになります。

   ```
   Username: debug
   Password: bEDp3MOHdj8ou2w5de_sCBI2XAaguy3a8XxREXAMPLE
   Password will expire at: 2021-04-01T17:01:43.921999931-07:00
   ```

   デバッグビューコンポーネントは 4 時間続くセッションを作成し、さらにローカルデバッグコンソールを再度確認するため、新しいパスワードを生成する必要があります。

3. コアデバイスで `http://localhost:1441` を開きます。または、SSH 経由でポートを転送した場合、開発コンピュータで開きます。

4. `get-debug-password` コマンドが以前に出力したユーザー名とパスワードを使用して、ウェブサイトにサインインします。

   ローカルデバッグコンソールが開きます。

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.4.1	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.4.0	新機能 • ストリームマネージャーのデバッグコンソールを追加します。

バージョン	変更
2.3.1	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.3.0	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。 新機能 • PubSub および AWS IoT Core MQTT デバッグクライアントが含まれます。
2.2.7	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.2.6	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.2.5	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.2.4	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.2.3	バグ修正と機能向上 • コンポーネントが SSL プライベートキーを保持しているキーストアを復号化できなかったときに起動できない問題を修正します。 • Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.2.2	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.2.1	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.2.0	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.1.0	新機能 • HTTPS を使用して、ローカルデバッグコンソールへの接続を保護します。HTTPS はデフォルトで有効になっています。 バグ修正と機能向上 • 設定エディタのフラッシュバーメッセージを消去できます。
2.0.3	当初のバージョン

ログマネージャー

ログマネージャーコンポーネント (`aws.greengrass.LogManager`) は、AWS IoT Greengrassコアデバイスから Amazon CloudWatch Logs にログをアップロードします。Greengrass nucleus、他の Greengrass コンポーネント、Greengrass コンポーネントではない他のアプリケーションとサービスからログをアップロードできます。Logs およびローカルファイルシステムで CloudWatch ログをモニタリングする方法の詳細については、「」を参照してくださいAWS IoT Greengrass ログのモニタリング。

ログマネージャーコンポーネントを使用して CloudWatch Logs に書き込む場合は、次の考慮事項が適用されます。

- ログ遅延

  > **Note**
  >
  > ログマネージャーバージョン 2.3.0 にアップグレードすることをお勧めします。これにより、ローテーションされたアクティブなログファイルのログ遅延が軽減されます。ログマネージャー 2.3.0 にアップグレードする際には、Greengrass nucleus 2.9.1 にもアップグレードすることをお勧めします。

  ログマネージャーコンポーネントバージョン 2.2.8 (およびそれ以前) は、ローテーションされたログファイルからのみ、ログを処理およびアップロードします。デフォルトで、AWS IoT Greengrass Core ソフトウェアは、1 時間ごと、または 1,024 KB に達した後に、ログファイルをローテーションします。その結果、ログマネージャーコンポーネントは、AWS IoT Greengrass Core ソフトウェアまたは Greengrass コンポーネントが 1,024 KB を超えるログを書き込んだ後に

のみ、ログをアップロードします。ログファイルの容量上限を低く設定して、ログファイルのローテーション頻度を上げることができます。これにより、ログマネージャーコンポーネントはログをより頻繁に CloudWatch Logs にアップロードします。

ログマネージャーコンポーネントバージョン 2.3.0 (およびそれ以降) は、すべてのログを処理およびアップロードします。新しいログを書き込むと、ログマネージャーバージョン 2.3.0 (およびそれ以降) は、ローテーションを待つことなく、そのアクティブなログファイルを処理して直接アップロードします。そのため、5 分以内に新しいログを表示することができます。

ログマネージャーコンポーネントは、新しいログを定期的にアップロードします。デフォルトで、ログマネージャーコンポーネントは 5 分ごとに新しいログをアップロードします。ログマネージャーコンポーネントが を設定することでログをより頻繁に CloudWatch Logs にアップロードするように、アップロード間隔を短く設定できます`periodicUploadIntervalSec`。この定期的な間隔を設定する方法の詳細については、「設定」を参照してください。

ログは、同じ Greengrass ファイルシステムからほぼリアルタイムでアップロードできます。ログをリアルタイムで監視する必要がある場合、ファイルシステムログの使用を検討してください。

> **Note**
>
> ログの書き込み先として別のファイルシステムを使用している場合、ログマネージャーは、ログマネージャーコンポーネントバージョン 2.2.8 以前の動作に戻ります。ファイルシステムログへのアクセスについては、「ファイルシステムログをアクセス」を参照してください。

- クロックスキュー

  ログマネージャーコンポーネントは、標準の Signature Version 4 署名プロセスを使用して、CloudWatch ログへの API リクエストを作成します。コアデバイスのシステム時間が同期していない時間が 15 分を超えると、 CloudWatch Logs はリクエストを拒否します。詳細については、「AWS 全般のリファレンス」の「署名バージョン 4 の署名プロセス」を参照してください。

このコンポーネントがログをアップロードするロググループとログストリームの情報については、「使用方法」を参照してください。

トピック

- バージョン
- タイプ

- オペレーティングシステム
- 要件
- 依存関係
- 構成
- 使用方法
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

タイプ

このコンポーネントはプラグインコンポーネント (`aws.greengrass.plugin`) です。Greengrass nucleus は、nucleus と同じ Java バーチャルマシン (JVM) でこのコンポーネントを実行します。コアデバイスでこのコンポーネントのバージョンを変更するとき、nucleus が再起動します。

このコンポーネントは、Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- 次の IAM ポリシーの例で示されているように、Greengrass デバイスのロールは、`logs:CreateLogGroup`、`logs:CreateLogStream`、`logs:PutLogEvents`、`logs:Describe` アクションを許可する必要があります。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws:logs:*:*:*"
    }
  ]
}
```

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアをインストール時に作成する Greengrass デバイスのロールには、デフォルトでこのポリシーの例の許可が含まれています。

詳細については、「Amazon Logs ユーザーガイド」の CloudWatch 「ログにアイデンティティベースのポリシー (IAM ポリシー) を使用する」を参照してください。 CloudWatch

- ログマネージャーコンポーネントは、VPC での実行がサポートされています。このコンポーネントを VPC にデプロイするには、以下が必要です。
  - ログマネージャーコンポーネントには、 `logs.region.amazonaws.com` の VPC エンドポイントを持つ への接続が必要です`com.amazonaws.us-east-1.logs`。

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
`logs.region.amazonaws.com`	443	いいえ	CloudWatch Logs にログを書き込む場合に必要です。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.3.7

次の表に、このコンポーネントのバージョン 2.3.7 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.13.0	ソフト

2.3.5 and 2.3.6

次の表に、このコンポーネントのバージョン 2.3.5 および 2.3.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.12.0	ソフト

2.3.3 – 2.3.4

次の表に、このコンポーネントのバージョン 2.3.3 から 2.3.4 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.11.0	ソフト

2.2.8 – 2.3.2

次の表に、このコンポーネントのバージョン 2.2.8 から 2.3.2 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.10.0	ソフト

2.2.7

次の表に、このコンポーネントのバージョン 2.2.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.9.0	ソフト

2.2.6

次の表に、このコンポーネントのバージョン 2.2.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.8.0	ソフト

2.2.5

次の表に、このコンポーネントのバージョン 2.2.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.7.0	ソフト

2.2.1 - 2.2.4

次の表に、このコンポーネントのバージョン 2.2.1～2.2.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.6.0	ソフト

2.1.3 and 2.2.0

次の表に、このコンポーネントのバージョン 2.1.3 および 2.2.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.5.0	ソフト

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.4.0	ソフト

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.3.0	ソフト

2.1.0

次の表に、このコンポーネントのバージョン 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.1.0 <2.2.0	ソフト

2.0.x

次の表に、このコンポーネントのバージョン 2.0.x の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.3 <2.1.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

v2.3.6 – v2.3.7

`logsUploaderConfiguration`

(オプション) ログマネージャーコンポーネントがアップロードするログの設定。このオブジェクトには、次の情報が含まれます。

`systemLogsConfiguration`

(オプション) Greengrass nucleus とプラグインコンポーネントのログを含む AWS IoT Greengrass Core ソフトウェアシステムログの設定。ログマネージャーコンポーネントが

システムログを管理できるようにするため、この設定を指定します。このオブジェクトには、次の情報が含まれます。

`uploadToCloudWatch`

(オプション) システムログを Logs CloudWatch にアップロードできます。

デフォルト: `false`

`minimumLogLevel`

(オプション) アップロードするログメッセージの最小レベル。この最小レベルは、Greengrass nucleus コンポーネントが JSON 形式のログを出力するように設定した場合のみ、適用されます。JSON 形式のログを有効にするには、ログ形式パラメータ (`logging.format`) の `JSON` を指定します。

こちらにレベル順に一覧表示されているログレベルから選択します。

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

デフォルト: `INFO`

`diskSpaceLimit`

(オプション) Greengrass システムログファイルの最大合計サイズ (`diskSpaceLimitUnit` で指定する単位)。Greengrass システムログファイルの合計サイズがこの最大合計サイズを超えると、AWS IoT Greengrass Core ソフトウェアは最も古い Greengrass システムログファイルを削除します。

このパラメータは、Greengrass nucleus コンポーネントのログサイズ上限パラメータ (`totalLogsSizeKB`) と同等です。AWS IoT Greengrass Core ソフトウェアは、Greengrass システムログの最大合計サイズとして 2 つの値の最小値を使用します。

`diskSpaceLimitUnit`

(オプション) `diskSpaceLimit` の単位。次のオプションから選択します。

- `KB` - キロバイト

- MB - メガバイト
- GB - ギガバイト

デフォルト: KB

`deleteLogFileAfterCloudUpload`

(オプション) ログマネージャーコンポーネントがログを CloudWatch Logs にアップロードした後、ログファイルを削除できます。

デフォルト: `false`

`componentLogsConfigurationMap`

(オプション) コアデバイスのコンポーネント用ログ設定のマップ。このマップにある各 `componentName` オブジェクトは、コンポーネントまたはアプリケーションのログ設定を定義します。ログマネージャーコンポーネントは、これらのコンポーネントログを CloudWatch ログにアップロードします。

> **Important**
>
> コンポーネントごとに 1 つの設定キーを使用することを強くお勧めします。`logFileRegex` を使用するときは、アクティブに書き込まれているログファイルが 1 つしかないファイルのグループのみに的を絞ってください。この推奨事項に従わないと、重複したログが にアップロードされる可能性があります CloudWatch。単一の正規表現で複数のアクティブなログファイルを対象とする場合は、ログマネージャーを v2.3.1 以降にアップグレードし、設定例を使用した設定変更のご検討をお勧めします。

> **Note**
>
> v2.2.0 より前のバージョンのログマネージャーからアップグレードする場合、`componentLogsConfigurationMap` ではなく、`componentLogsConfiguration` リストを使い続けることができます。ただし、マージとリセットの更新を使用して特定コンポーネントの設定を修正できるように、マップ形式を使用することを強くお勧めします。`componentLogsConfiguration` パラメータの情報については、このコンポーネントの v2.1.x の設定パラメータを参照してください。

componentName

componentName コンポーネントのログ設定またはこのログ設定のアプリケーション。Greengrass コンポーネントの名前または別の値を指定して、このロググループを識別できます。

各オブジェクトには、次の情報が含まれます:

`minimumLogLevel`

(オプション) アップロードするログメッセージの最小レベル。この最小レベルは、このコンポーネントのログが特定の JSON 形式を使用している場合にのみ適用されます。JSON 形式は、 のAWS IoT Greengrassログ記録モジュールリポジトリにあります GitHub。

こちらにレベル順に一覧表示されているログレベルから選択します。

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

デフォルト: `INFO`

`diskSpaceLimit`

(オプション) このコンポーネントにおけるすべてのログファイルの最大合計サイズ (`diskSpaceLimitUnit` で指定する単位)。このコンポーネントのログファイルの合計サイズがこの最大合計サイズを超えると、AWS IoT Greengrass Core ソフトウェアはこのコンポーネントの最も古いログファイルを削除します。

このパラメータは、Greengrass nucleus コンポーネントのログサイズ上限パラメータ (`totalLogsSizeKB`) に関連しています。AWS IoT Greengrass Core ソフトウェアは、このコンポーネントの最大合計ログサイズとして 2 つの値の最小値を使用します。

`diskSpaceLimitUnit`

(オプション) `diskSpaceLimit` の単位。次のオプションから選択します。

- `KB` - キロバイト

- MB - メガバイト
- GB - ギガバイト

デフォルト: KB

`logFileDirectoryPath`

(オプション) このコンポーネントのログファイルを含むフォルダへのパス。

標準出力 (stdout) と標準エラー (stderr) に出力する Greengrass コンポーネントには、このパラメータを指定する必要はありません。

デフォルト: */greengrass/v2*/logs。

`logFileRegex`

(オプション) コンポーネントまたはアプリケーションが使用するログファイル名の形式を指定する正規表現。ログマネージャーコンポーネントは、この正規表現を使用して `logFileDirectoryPath` のフォルダのログファイルを識別します。

標準出力 (stdout) と標準エラー (stderr) に出力する Greengrass コンポーネントには、このパラメータを指定する必要はありません。

コンポーネントまたはアプリケーションがログファイルをローテーションする場合、ローテーションされたログファイル名と一致する正規表現を指定します。例えば、**`hello_world\\\\w*.log`** が Hello World アプリケーションにログをアップロードするように指定します。`\\\\w*` パターンは、英数字とアンダースコアを含むゼロ以上の単語文字と一致します。この正規表現は、名前のタイムスタンプの有無にかかわらず、ログファイルを照合します。この例では、ログマネージャーは次のログファイルをアップロードします。

- `hello_world.log` - Hello World アプリケーションの最新ログファイル。
- `hello_world_2020_12_15_17_0.log` - Hello World アプリケーションの古いログファイル。

デフォルト: *componentName* がこのログ設定におけるコンポーネントの名前である場合の *componentName*`\\\\w*.log`。

`deleteLogFileAfterCloudUpload`

(オプション) ログマネージャーコンポーネントがログを CloudWatch Logs にアップロードした後、ログファイルを削除できます。

デフォルト: false

multiLineStartPattern

(オプション) 新しい行のログメッセージが新しいログメッセージであるときに識別する正規表現。正規表現が新しい行と一致しない場合、ログマネージャーコンポーネントは新しい行を前の行のログメッセージに追加します。

デフォルトでは、ログマネージャーコンポーネントは、行が空白文字 (タブまたはスペースなど) で始まるかどうかについてチェックします。そうでない場合、ログマネージャーはその行を新しいログメッセージとして処理します。それ以外の場合、その行を現在のログメッセージに追加します。この動作により、ログマネージャーコンポーネントが複数行 (スタックトレースなど) にまたがるメッセージを分割しないようにします。

periodicUploadIntervalSec

(オプション) ログマネージャーコンポーネントがアップロードする新しいログファイルをチェックする期間 (秒)。

デフォルト: 300 (5 分)

最小: 0.000001 (1 マイクロ秒)

deprecatedVersionSupport

ログマネージャーが v2.3.5 で導入された口ギング速度の向上を使用するかどうかを示します。値を false に設定してこの機能強化を使用します。

ログマネージャー v2.3.1 以前からアップグレードするときにこの値を false に設定すると、重複したログエントリがアップロードされる場合があります。

デフォルトは true です。

Example 例: 設定マージの更新

次の設定例では、システムログとcom.example.HelloWorldコンポーネントログを CloudWatch ログにアップロードするように指定しています。

```
{
  "logsUploaderConfiguration": {
```

```
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true",
      "minimumLogLevel": "INFO",
      "diskSpaceLimit": "10",
      "diskSpaceLimitUnit": "MB",
      "deleteLogFileAfterCloudUpload": "false"
    },
    "componentLogsConfigurationMap": {
      "com.example.HelloWorld": {
        "minimumLogLevel": "INFO",
        "diskSpaceLimit": "20",
        "diskSpaceLimitUnit": "MB",
        "deleteLogFileAfterCloudUpload": "false"
      }
    }
  },
  "periodicUploadIntervalSec": "300",
  "deprecatedVersionSupport": "false"
}
```

Example 例: ログマネージャー v2.3.1 を使用して、複数のアクティブなログファイルをアップロードための設定

複数のアクティブなログファイルを対象にする場合は、次の設定例が推奨されます。この設定例では、 にアップロードするアクティブなログファイルを指定します CloudWatch。この設定例を使用すると、logFileRegexに一致するローテーションファイルもアップロードされます。この設定例は、ログマネージャー v2.3.1 でサポートされています。

```
{
  "logsUploaderConfiguration": {
    "componentLogsConfigurationMap": {
      "com.example.A": {
        "logFileRegex": "com.example.A\\w*.log",
        "deleteLogFileAfterCloudUpload": "false"
      }
      "com.example.B": {
        "logFileRegex": "com.example.B\\w*.log",
        "deleteLogFileAfterCloudUpload": "false"
      }
    }
  },
  "periodicUploadIntervalSec": "10"
```

```
}
```

v2.3.x

`logsUploaderConfiguration`

(オプション) ログマネージャーコンポーネントがアップロードするログの設定。このオブジェクトには、次の情報が含まれます。

`systemLogsConfiguration`

(オプション) Greengrass nucleus とプラグインコンポーネントのログを含む AWS IoT Greengrass Core ソフトウェアシステムログの設定。ログマネージャーコンポーネントがシステムログを管理できるようにするため、この設定を指定します。このオブジェクトには、次の情報が含まれます。

`uploadToCloudWatch`

(オプション) システムログを Logs CloudWatch にアップロードできます。

デフォルト: `false`

`minimumLogLevel`

(オプション) アップロードするログメッセージの最小レベル。この最小レベルは、Greengrass nucleus コンポーネントが JSON 形式のログを出力するように設定した場合のみ、適用されます。JSON 形式のログを有効にするには、ログ形式パラメータ (`logging.format`) の `JSON` を指定します。

こちらにレベル順に一覧表示されているログレベルから選択します。

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

デフォルト: `INFO`

`diskSpaceLimit`

(オプション) Greengrass システムログファイルの最大合計サイズ (`diskSpaceLimitUnit` で指定する単位)。Greengrass システムログファイルの合計

サイズがこの最大合計サイズを超えると、AWS IoT Greengrass Core ソフトウェアは最も古い Greengrass システムログファイルを削除します。

このパラメータは、Greengrass nucleus コンポーネントのログサイズ上限パラメータ (`totalLogsSizeKB`) と同等です。AWS IoT Greengrass Core ソフトウェアは、Greengrass システムログの最大合計サイズとして 2 つの値の最小値を使用します。

`diskSpaceLimitUnit`

(オプション) `diskSpaceLimit` の単位。次のオプションから選択します。

- `KB` - キロバイト
- `MB` - メガバイト
- `GB` - ギガバイト

デフォルト: `KB`

`deleteLogFileAfterCloudUpload`

(オプション) ログマネージャーコンポーネントがログを CloudWatch Logs にアップロードした後、ログファイルを削除できます。

デフォルト: `false`

`componentLogsConfigurationMap`

(オプション) コアデバイスのコンポーネント用ログ設定のマップ。このマップにある各 `componentName` オブジェクトは、コンポーネントまたはアプリケーションのログ設定を定義します。ログマネージャーコンポーネントは、これらのコンポーネントログを CloudWatch ログにアップロードします。

> **Important**
>
> コンポーネントごとに 1 つの設定キーを使用することを強くお勧めします。`logFileRegex` を使用するときは、アクティブに書き込まれているログファイルが 1 つしかないファイルのグループのみに的を絞ってください。この推奨事項に従わないと、重複したログが にアップロードされる可能性があります CloudWatch。単一の正規表現で複数のアクティブなログファイルを対象とする場合は、ログマネージャーを v2.3.1 にアップグレードし、設定例を使用した設定変更のご検討をお勧めします。

> **Note**
>
> v2.2.0 より前のバージョンのログマネージャーからアップグレードする場合、componentLogsConfigurationMap ではなく、componentLogsConfiguration リストを使い続けることができます。ただし、マージとリセットの更新を使用して特定コンポーネントの設定を修正できるように、マップ形式を使用することを強くお勧めします。componentLogsConfiguration パラメータの情報については、このコンポーネントの v2.1.x の設定パラメータを参照してください。

componentName

componentName コンポーネントのログ設定またはこのログ設定のアプリケーション。Greengrass コンポーネントの名前または別の値を指定して、このロググループを識別できます。

各オブジェクトには、次の情報が含まれます:

minimumLogLevel

(オプション) アップロードするログメッセージの最小レベル。この最小レベルは、このコンポーネントのログが特定の JSON 形式を使用している場合にのみ適用されます。JSON 形式は、 のAWS IoT Greengrassログ記録モジュールリポジトリにあります GitHub。

こちらにレベル順に一覧表示されているログレベルから選択します。

- DEBUG
- INFO
- WARN
- ERROR

デフォルト: INFO

diskSpaceLimit

(オプション) このコンポーネントにおけるすべてのログファイルの最大合計サイズ (diskSpaceLimitUnit で指定する単位)。このコンポーネントのログファイルの

合計サイズがこの最大合計サイズを超えると、AWS IoT Greengrass Core ソフトウェアはこのコンポーネントの最も古いログファイルを削除します。

このパラメータは、Greengrass nucleus コンポーネントのログサイズ上限パラメータ (`totalLogsSizeKB`) に関連しています。AWS IoT Greengrass Core ソフトウェアは、このコンポーネントの最大合計ログサイズとして 2 つの値の最小値を使用します。

`diskSpaceLimitUnit`

(オプション) `diskSpaceLimit` の単位。次のオプションから選択します。

- `KB` - キロバイト
- `MB` - メガバイト
- `GB` - ギガバイト

デフォルト: `KB`

`logFileDirectoryPath`

(オプション) このコンポーネントのログファイルを含むフォルダへのパス。

標準出力 (stdout) と標準エラー (stderr) に出力する Greengrass コンポーネントには、このパラメータを指定する必要はありません。

デフォルト: */greengrass/v2*/logs。

`logFileRegex`

(オプション) コンポーネントまたはアプリケーションが使用するログファイル名の形式を指定する正規表現。ログマネージャーコンポーネントは、この正規表現を使用して `logFileDirectoryPath` のフォルダのログファイルを識別します。

標準出力 (stdout) と標準エラー (stderr) に出力する Greengrass コンポーネントには、このパラメータを指定する必要はありません。

コンポーネントまたはアプリケーションがログファイルをローテーションする場合、ローテーションされたログファイル名と一致する正規表現を指定します。例えば、**`hello_world\\\\w*.log`** が Hello World アプリケーションにログをアップロードするように指定します。`\\\\w*` パターンは、英数字とアンダースコアを含むゼロ以上の単語文字と一致します。この正規表現は、名前のタイムスタンプの有

無にかかわらず、ログファイルを照合します。この例では、ログマネージャーは次のログファイルをアップロードします。

- `hello_world.log` - Hello World アプリケーションの最新ログファイル。
- `hello_world_2020_12_15_17_0.log` - Hello World アプリケーションの古いログファイル。

デフォルト: *componentName* がこのログ設定におけるコンポーネントの名前である場合の *componentName*`\\\\w*.log`。

`deleteLogFileAfterCloudUpload`

(オプション) ログマネージャーコンポーネントがログを CloudWatch Logs にアップロードした後、ログファイルを削除できます。

デフォルト: `false`

`multiLineStartPattern`

(オプション) 新しい行のログメッセージが新しいログメッセージであるときに識別する正規表現。正規表現が新しい行と一致しない場合、ログマネージャーコンポーネントは新しい行を前の行のログメッセージに追加します。

デフォルトでは、ログマネージャーコンポーネントは、行が空白文字 (タブまたはスペースなど) で始まるかどうかについてチェックします。そうでない場合、ログマネージャーはその行を新しいログメッセージとして処理します。それ以外の場合、その行を現在のログメッセージに追加します。この動作により、ログマネージャーコンポーネントが複数行 (スタックトレースなど) にまたがるメッセージを分割しないようにします。

`periodicUploadIntervalSec`

(オプション) ログマネージャーコンポーネントがアップロードする新しいログファイルをチェックする期間 (秒)。

デフォルト: `300` (5 分)

最小: `0.000001` (1 マイクロ秒)

Example 例: 設定マージの更新

次の設定例では、システムログと`com.example.HelloWorld`コンポーネントログを CloudWatch ログにアップロードするように指定しています。

```
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true",
      "minimumLogLevel": "INFO",
      "diskSpaceLimit": "10",
      "diskSpaceLimitUnit": "MB",
      "deleteLogFileAfterCloudUpload": "false"
    },
    "componentLogsConfigurationMap": {
      "com.example.HelloWorld": {
        "minimumLogLevel": "INFO",
        "diskSpaceLimit": "20",
        "diskSpaceLimitUnit": "MB",
        "deleteLogFileAfterCloudUpload": "false"
      }
    }
  },
  "periodicUploadIntervalSec": "300"
}
```

Example 例: ログマネージャー v2.3.1 を使用して、複数のアクティブなログファイルをアップロードための設定

複数のアクティブなログファイルを対象にする場合は、次の設定例が推奨されます。この設定例では、 にアップロードするアクティブなログファイルを指定します CloudWatch。この設定例を使用すると、`logFileRegex`に一致するローテーションファイルもアップロードされます。この設定例は、ログマネージャー v2.3.1 でサポートされています。

```
{
  "logsUploaderConfiguration": {
    "componentLogsConfigurationMap": {
      "com.example.A": {
        "logFileRegex": "com.example.A\\w*.log",
        "deleteLogFileAfterCloudUpload": "false"
      }
      "com.example.B": {
        "logFileRegex": "com.example.B\\w*.log",
        "deleteLogFileAfterCloudUpload": "false"
      }
    }
  },
```

```
  "periodicUploadIntervalSec": "10"
}
```

v2.2.x

`logsUploaderConfiguration`

(オプション) ログマネージャーコンポーネントがアップロードするログの設定。このオブジェクトには、次の情報が含まれます。

`systemLogsConfiguration`

(オプション) Greengrass nucleus とプラグインコンポーネントのログを含む AWS IoT Greengrass Core ソフトウェアシステムログの設定。ログマネージャーコンポーネントがシステムログを管理できるようにするため、この設定を指定します。このオブジェクトには、次の情報が含まれます。

`uploadToCloudWatch`

(オプション) システムログを Logs CloudWatch にアップロードできます。

デフォルト: `false`

`minimumLogLevel`

(オプション) アップロードするログメッセージの最小レベル。この最小レベルは、Greengrass nucleus コンポーネントが JSON 形式のログを出力するように設定した場合のみ、適用されます。JSON 形式のログを有効にするには、ログ形式パラメータ (`logging.format`) の `JSON` を指定します。

こちらにレベル順に一覧表示されているログレベルから選択します。

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

デフォルト: `INFO`

`diskSpaceLimit`

(オプション) Greengrass システムログファイルの最大合計サイズ (`diskSpaceLimitUnit` で指定する単位)。Greengrass システムログファイルの合計

サイズがこの最大合計サイズを超えると、AWS IoT Greengrass Core ソフトウェアは最も古い Greengrass システムログファイルを削除します。

このパラメータは、Greengrass nucleus コンポーネントのログサイズ上限パラメータ (`totalLogsSizeKB`) と同等です。AWS IoT Greengrass Core ソフトウェアは、Greengrass システムログの最大合計サイズとして 2 つの値の最小値を使用します。

`diskSpaceLimitUnit`

(オプション) `diskSpaceLimit` の単位。次のオプションから選択します。

- `KB` - キロバイト
- `MB` - メガバイト
- `GB` - ギガバイト

デフォルト: `KB`

`deleteLogFileAfterCloudUpload`

(オプション) ログマネージャーコンポーネントがログを CloudWatch Logs にアップロードした後、ログファイルを削除できます。

デフォルト: `false`

`componentLogsConfigurationMap`

(オプション) コアデバイスのコンポーネント用ログ設定のマップ。このマップにある各 `componentName` オブジェクトは、コンポーネントまたはアプリケーションのログ設定を定義します。ログマネージャーコンポーネントは、これらのコンポーネントログを CloudWatch ログにアップロードします。

> **Note**
>
> v2.2.0 より前のバージョンのログマネージャーからアップグレードする場合、`componentLogsConfigurationMap` ではなく、`componentLogsConfiguration` リストを使い続けることができます。ただし、マージとリセットの更新を使用して特定コンポーネントの設定を修正できるように、マップ形式を使用することを強くお勧めします。`componentLogsConfiguration` パラメータの情報については、このコンポーネントの v2.1.x の設定パラメータを参照してください。

componentName

componentName コンポーネントのログ設定またはこのログ設定のアプリケーション。Greengrass コンポーネントの名前または別の値を指定して、このロググループを識別できます。

各オブジェクトには、次の情報が含まれます:

`minimumLogLevel`

(オプション) アップロードするログメッセージの最小レベル。この最小レベルは、このコンポーネントのログが特定の JSON 形式を使用している場合にのみ適用されます。JSON 形式は、 のAWS IoT Greengrassログ記録モジュールリポジトリにあります GitHub。

こちらにレベル順に一覧表示されているログレベルから選択します。

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

デフォルト: `INFO`

`diskSpaceLimit`

(オプション) このコンポーネントにおけるすべてのログファイルの最大合計サイズ (`diskSpaceLimitUnit` で指定する単位)。このコンポーネントのログファイルの合計サイズがこの最大合計サイズを超えると、AWS IoT Greengrass Core ソフトウェアはこのコンポーネントの最も古いログファイルを削除します。

このパラメータは、Greengrass nucleus コンポーネントのログサイズ上限パラメータ (`totalLogsSizeKB`) に関連しています。AWS IoT Greengrass Core ソフトウェアは、このコンポーネントの最大合計ログサイズとして 2 つの値の最小値を使用します。

`diskSpaceLimitUnit`

(オプション) `diskSpaceLimit` の単位。次のオプションから選択します。

- `KB` - キロバイト
- `MB` - メガバイト

- GB - ギガバイト

デフォルト: KB

`logFileDirectoryPath`

(オプション) このコンポーネントのログファイルを含むフォルダへのパス。

標準出力 (stdout) と標準エラー (stderr) に出力する Greengrass コンポーネントには、このパラメータを指定する必要はありません。

デフォルト: */greengrass/v2*/logs。

`logFileRegex`

(オプション) コンポーネントまたはアプリケーションが使用するログファイル名の形式を指定する正規表現。ログマネージャーコンポーネントは、この正規表現を使用して `logFileDirectoryPath` のフォルダのログファイルを識別します。

標準出力 (stdout) と標準エラー (stderr) に出力する Greengrass コンポーネントには、このパラメータを指定する必要はありません。

コンポーネントまたはアプリケーションがログファイルをローテーションする場合、ローテーションされたログファイル名と一致する正規表現を指定します。例えば、**`hello_world\\\\w*.log`** が Hello World アプリケーションにログをアップロードするように指定します。`\\\\w*` パターンは、英数字とアンダースコアを含むゼロ以上の単語文字と一致します。この正規表現は、名前のタイムスタンプの有無にかかわらず、ログファイルを照合します。この例では、ログマネージャーは次のログファイルをアップロードします。

- `hello_world.log` - Hello World アプリケーションの最新ログファイル。
- `hello_world_2020_12_15_17_0.log` - Hello World アプリケーションの古いログファイル。

デフォルト: *componentName* がこのログ設定におけるコンポーネントの名前である場合の *componentName*`\\\\w*.log`。

`deleteLogFileAfterCloudUpload`

(オプション) ログマネージャーコンポーネントがログを CloudWatch Logs にアップロードした後、ログファイルを削除できます。

デフォルト: `false`

`multiLineStartPattern`

(オプション) 新しい行のログメッセージが新しいログメッセージであるときに識別する正規表現。正規表現が新しい行と一致しない場合、ログマネージャーコンポーネントは新しい行を前の行のログメッセージに追加します。

デフォルトでは、ログマネージャーコンポーネントは、行が空白文字 (タブまたはスペースなど) で始まるかどうかについてチェックします。そうでない場合、ログマネージャーはその行を新しいログメッセージとして処理します。それ以外の場合、その行を現在のログメッセージに追加します。この動作により、ログマネージャーコンポーネントが複数行 (スタックトレースなど) にまたがるメッセージを分割しないようにします。

`periodicUploadIntervalSec`

(オプション) ログマネージャーコンポーネントがアップロードする新しいログファイルをチェックする期間 (秒)。

デフォルト: `300` (5 分)

最小: `0.000001` (1 マイクロ秒)

Example 例: 設定マージの更新

次の設定例では、システムログと`com.example.HelloWorld`コンポーネントログを CloudWatch ログにアップロードするように指定しています。

```
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true",
      "minimumLogLevel": "INFO",
      "diskSpaceLimit": "10",
      "diskSpaceLimitUnit": "MB",
      "deleteLogFileAfterCloudUpload": "false"
    },
    "componentLogsConfigurationMap": {
      "com.example.HelloWorld": {
        "minimumLogLevel": "INFO",
        "diskSpaceLimit": "20",
        "diskSpaceLimitUnit": "MB",
```

```
        "deleteLogFileAfterCloudUpload": "false"
      }
    }
  },
  "periodicUploadIntervalSec": "300"
}
```

v2.1.x

`logsUploaderConfiguration`

(オプション) ログマネージャーコンポーネントがアップロードするログの設定。このオブジェクトには、次の情報が含まれます。

`systemLogsConfiguration`

(オプション) Greengrass nucleus とプラグインコンポーネントのログを含む AWS IoT Greengrass Core ソフトウェアシステムログの設定。ログマネージャーコンポーネントがシステムログを管理できるようにするため、この設定を指定します。このオブジェクトには、次の情報が含まれます。

`uploadToCloudWatch`

(オプション) システムログを Logs CloudWatch にアップロードできます。

デフォルト: `false`

`minimumLogLevel`

(オプション) アップロードするログメッセージの最小レベル。この最小レベルは、Greengrass nucleus コンポーネントが JSON 形式のログを出力するように設定した場合のみ、適用されます。JSON 形式のログを有効にするには、ログ形式パラメータ (`logging.format`) の `JSON` を指定します。

こちらにレベル順に一覧表示されているログレベルから選択します。

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

デフォルト: `INFO`

`diskSpaceLimit`

(オプション) Greengrass システムログファイルの最大合計サイズ (`diskSpaceLimitUnit` で指定する単位)。Greengrass システムログファイルの合計サイズがこの最大合計サイズを超えると、AWS IoT Greengrass Core ソフトウェアは最も古い Greengrass システムログファイルを削除します。

このパラメータは、Greengrass nucleus コンポーネントのログサイズ上限パラメータ (`totalLogsSizeKB`) と同等です。AWS IoT Greengrass Core ソフトウェアは、Greengrass システムログの最大合計サイズとして 2 つの値の最小値を使用します。

`diskSpaceLimitUnit`

(オプション) `diskSpaceLimit` の単位。次のオプションから選択します。

- `KB` - キロバイト
- `MB` - メガバイト
- `GB` - ギガバイト

デフォルト: `KB`

`deleteLogFileAfterCloudUpload`

(オプション) ログマネージャーコンポーネントがログを CloudWatch Logs にアップロードした後、ログファイルを削除できます。

デフォルト: `false`

`componentLogsConfiguration`

(オプション) コアデバイスのコンポーネント用のログ設定リスト。このリストに記載された各設定内容は、コンポーネントまたはアプリケーションのログ設定を定義します。ログマネージャーコンポーネントは、これらのコンポーネントログを CloudWatch Logs にアップロードします。

各オブジェクトには、次の情報が含まれます:

`componentName`

このログ設定のコンポーネントまたはアプリケーションの名前。Greengrass コンポーネントの名前または別の値を指定して、このロググループを識別できます。

`minimumLogLevel`

(オプション) アップロードするログメッセージの最小レベル。この最小レベルは、このコンポーネントのログが特定の JSON 形式を使用している場合にのみ適用されます。JSON 形式は、 のAWS IoT Greengrassログ記録モジュールリポジトリにありますGitHub。

こちらにレベル順に一覧表示されているログレベルから選択します。

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

デフォルト: `INFO`

`diskSpaceLimit`

(オプション) このコンポーネントにおけるすべてのログファイルの最大合計サイズ (`diskSpaceLimitUnit` で指定する単位)。このコンポーネントのログファイルの合計サイズがこの最大合計サイズを超えると、AWS IoT Greengrass Core ソフトウェアはこのコンポーネントの最も古いログファイルを削除します。

このパラメータは、Greengrass nucleus コンポーネントのログサイズ上限パラメータ (`totalLogsSizeKB`) に関連しています。AWS IoT Greengrass Core ソフトウェアは、このコンポーネントの最大合計ログサイズとして 2 つの値の最小値を使用します。

`diskSpaceLimitUnit`

(オプション) `diskSpaceLimit` の単位。次のオプションから選択します。

- `KB` - キロバイト
- `MB` - メガバイト
- `GB` - ギガバイト

デフォルト: `KB`

`logFileDirectoryPath`

(オプション) このコンポーネントのログファイルを含むフォルダへのパス。

標準出力 (stdout) と標準エラー (stderr) に出力する Greengrass コンポーネントには、このパラメータを指定する必要はありません。

デフォルト: *`/greengrass/v2`*`/logs`。

`logFileRegex`

(オプション) コンポーネントまたはアプリケーションが使用するログファイル名の形式を指定する正規表現。ログマネージャーコンポーネントは、この正規表現を使用して `logFileDirectoryPath` のフォルダのログファイルを識別します。

標準出力 (stdout) と標準エラー (stderr) に出力する Greengrass コンポーネントには、このパラメータを指定する必要はありません。

コンポーネントまたはアプリケーションがログファイルをローテーションする場合、ローテーションされたログファイル名と一致する正規表現を指定します。例えば、**`hello_world\\\\w*.log`** が Hello World アプリケーションにログをアップロードするように指定します。`\\\\w*` パターンは、英数字とアンダースコアを含むゼロ以上の単語文字と一致します。この正規表現は、名前のタイムスタンプの有無にかかわらず、ログファイルを照合します。この例では、ログマネージャーは次のログファイルをアップロードします。

- `hello_world.log` - Hello World アプリケーションの最新ログファイル。
- `hello_world_2020_12_15_17_0.log` - Hello World アプリケーションの古いログファイル。

デフォルト: *`componentName`* がこのログ設定におけるコンポーネントの名前である場合の *`componentName`*`\\\\w*.log`。

`deleteLogFileAfterCloudUpload`

(オプション) ログマネージャーコンポーネントがログを CloudWatch Logs にアップロードした後、ログファイルを削除できます。

デフォルト: `false`

`multiLineStartPattern`

(オプション) 新しい行のログメッセージが新しいログメッセージであるときに識別する正規表現。正規表現が新しい行と一致しない場合、ログマネージャーコンポーネントは新しい行を前の行のログメッセージに追加します。

デフォルトでは、ログマネージャーコンポーネントは、行が空白文字 (タブまたはスペースなど) で始まるかどうかについてチェックします。そうでない場合、ログマネージャーはその行を新しいログメッセージとして処理します。それ以外の場合、その行を現在のログメッセージに追加します。この動作により、ログマネージャーコンポーネントが複数行 (スタックトレースなど) にまたがるメッセージを分割しないようにします。

`periodicUploadIntervalSec`

(オプション) ログマネージャーコンポーネントがアップロードする新しいログファイルをチェックする期間 (秒)。

デフォルト: `300` (5 分)

最小: `0.000001` (1 マイクロ秒)

Example 例: 設定マージの更新

次の設定例では、システムログと`com.example.HelloWorld`コンポーネントログを CloudWatch ログにアップロードするように指定しています。

```
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true",
      "minimumLogLevel": "INFO",
      "diskSpaceLimit": "10",
      "diskSpaceLimitUnit": "MB",
      "deleteLogFileAfterCloudUpload": "false"
    },
    "componentLogsConfiguration": [
      {
        "componentName": "com.example.HelloWorld",
        "minimumLogLevel": "INFO",
        "diskSpaceLimit": "20",
        "diskSpaceLimitUnit": "MB",
        "deleteLogFileAfterCloudUpload": "false"
      }
    ]
  },
  "periodicUploadIntervalSec": "300"
}
```

v2.0.x

`logsUploaderConfiguration`

(オプション) ログマネージャーコンポーネントがアップロードするログの設定。このオブジェクトには、次の情報が含まれます。

`systemLogsConfiguration`

(オプション) AWS IoT Greengrass Core ソフトウェアシステムログの設定。ログマネージャーコンポーネントがシステムログを管理できるようにするため、この設定を指定します。このオブジェクトには、次の情報が含まれます。

`uploadToCloudWatch`

(オプション) システムログを Logs CloudWatch にアップロードできます。

デフォルト: `false`

`minimumLogLevel`

(オプション) アップロードするログメッセージの最小レベル。この最小レベルは、Greengrass nucleus コンポーネントが JSON 形式のログを出力するように設定した場合のみ、適用されます。JSON 形式のログを有効にするには、ログ形式パラメータ (`logging.format`) の `JSON` を指定します。

こちらにレベル順に一覧表示されているログレベルから選択します。

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

デフォルト: `INFO`

`diskSpaceLimit`

(オプション) Greengrass システムログファイルの最大合計サイズ (`diskSpaceLimitUnit` で指定する単位)。Greengrass システムログファイルの合計サイズがこの最大合計サイズを超えると、AWS IoT Greengrass Core ソフトウェアは最も古い Greengrass システムログファイルを削除します。

このパラメータは、Greengrass nucleus コンポーネントのログサイズ上限パラメータ (`totalLogsSizeKB`) と同等です。AWS IoT Greengrass Core ソフトウェア

は、Greengrass システムログの最大合計サイズとして 2 つの値の最小値を使用します。

`diskSpaceLimitUnit`

(オプション) `diskSpaceLimit` の単位。次のオプションから選択します。

- `KB` - キロバイト
- `MB` - メガバイト
- `GB` - ギガバイト

デフォルト: `KB`

`deleteLogFileAfterCloudUpload`

(オプション) ログマネージャーコンポーネントがログを CloudWatch Logs にアップロードした後、ログファイルを削除できます。

デフォルト: `false`

`componentLogsConfiguration`

(オプション) コアデバイスのコンポーネント用のログ設定リスト。このリストに記載された各設定内容は、コンポーネントまたはアプリケーションのログ設定を定義します。ログマネージャーコンポーネントは、これらのコンポーネントログを CloudWatch Logs にアップロードします。

各オブジェクトには、次の情報が含まれます:

`componentName`

このログ設定のコンポーネントまたはアプリケーションの名前。Greengrass コンポーネントの名前または別の値を指定して、このロググループを識別できます。

`minimumLogLevel`

(オプション) アップロードするログメッセージの最小レベル。この最小レベルは、このコンポーネントのログが特定の JSON 形式を使用している場合にのみ適用されます。JSON 形式は、 のAWS IoT Greengrassログ記録モジュールリポジトリにありますGitHub。

こちらにレベル順に一覧表示されているログレベルから選択します。

- `DEBUG`
- `INFO`

- WARN
- ERROR

デフォルト: INFO

diskSpaceLimit

(オプション) このコンポーネントにおけるすべてのログファイルの最大合計サイズ (diskSpaceLimitUnit で指定する単位)。このコンポーネントのログファイルの合計サイズがこの最大合計サイズを超えると、AWS IoT Greengrass Core ソフトウェアはこのコンポーネントの最も古いログファイルを削除します。

このパラメータは、Greengrass nucleus コンポーネントのログサイズ上限パラメータ (totalLogsSizeKB) に関連しています。AWS IoT Greengrass Core ソフトウェアは、このコンポーネントの最大合計ログサイズとして 2 つの値の最小値を使用します。

diskSpaceLimitUnit

(オプション) diskSpaceLimit の単位。次のオプションから選択します。

- KB - キロバイト
- MB - メガバイト
- GB - ギガバイト

デフォルト: KB

logFileDirectoryPath

このコンポーネントのログファイルが含まれるフォルダへのパス。

Greengrass コンポーネントのログをアップロードするには、 を指定し***/greengrass/v2*****/logs**、 を Greengrass ルートフォルダ*/greengrass/v2*に置き換えます。

logFileRegex

コンポーネントまたはアプリケーションが使用するログファイル名の形式を指定する正規表現。ログマネージャーコンポーネントは、この正規表現を使用して logFileDirectoryPath のフォルダのログファイルを識別します。

Greengrass コンポーネントのログをアップロードするには、ローテーションされたログファイル名と一致する正規表現を指定します。例えば、**com.example.HelloWorld\\w*.log** が Hello World コンポーネント用にログをアップロードするように指定します。\\w* パターンは、英数字とアンダースコアを含

むゼロ以上の単語文字と一致します。この正規表現は、名前のタイムスタンプの有無にかかわらず、ログファイルを照合します。この例では、ログマネージャーは次のログファイルをアップロードします。

- `com.example.HelloWorld.log` - Hello World コンポーネントの最新ログファイル。
- `com.example.HelloWorld_2020_12_15_17_0.log` - Hello World コンポーネントのより古いログファイル。Greengrass nucleus は、ログファイルにローテーションするタイムスタンプを追加します。

`deleteLogFileAfterCloudUpload`

(オプション) ログマネージャーコンポーネントがログを CloudWatch Logs にアップロードした後、ログファイルを削除できます。

デフォルト: `false`

`multiLineStartPattern`

(オプション) 新しい行のログメッセージが新しいログメッセージであるときに識別する正規表現。正規表現が新しい行と一致しない場合、ログマネージャーコンポーネントは新しい行を前の行のログメッセージに追加します。

デフォルトでは、ログマネージャーコンポーネントは、行が空白文字 (タブまたはスペースなど) で始まるかどうかについてチェックします。そうでない場合、ログマネージャーはその行を新しいログメッセージとして処理します。それ以外の場合、その行を現在のログメッセージに追加します。この動作により、ログマネージャーコンポーネントが複数行 (スタックトレースなど) にまたがるメッセージを分割しないようにします。

`periodicUploadIntervalSec`

(オプション) ログマネージャーコンポーネントがアップロードする新しいログファイルをチェックする期間 (秒)。

デフォルト: `300` (5 分)

最小: `0.000001` (1 マイクロ秒)

Example 例: 設定マージの更新

次の設定例では、システムログと`com.example.HelloWorld`コンポーネントログを CloudWatch ログにアップロードするように指定しています。

```
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true",
      "minimumLogLevel": "INFO",
      "diskSpaceLimit": "10",
      "diskSpaceLimitUnit": "MB",
      "deleteLogFileAfterCloudUpload": "false"
    },
    "componentLogsConfiguration": [
      {
        "componentName": "com.example.HelloWorld",
        "minimumLogLevel": "INFO",
        "logFileDirectoryPath": "/greengrass/v2/logs",
        "logFileRegex": "com.example.HelloWorld\\w*.log",
        "diskSpaceLimit": "20",
        "diskSpaceLimitUnit": "MB",
        "deleteLogFileAfterCloudUpload": "false"
      }
    ]
  },
  "periodicUploadIntervalSec": "300"
}
```

使用方法

ログマネージャーコンポーネントは、次のロググループとログストリームにアップロードされます。

2.1.0 and later

ロググループ名

```
/aws/greengrass/componentType/region/componentName
```

ロググループ名は、次の変数を使用します。

- `componentType` - 次のいずれかに該当するコンポーネントのタイプ。
  - `GreengrassSystemComponent` — このロググループには、Greengrass nucleus と同じ JVM で実行される nucleus とプラグインコンポーネントのログが含まれます。コンポーネントは Greengrass nucleus の一部です。

- `UserComponent` – このロググループには、ジェネリックコンポーネント、Lambda コンポーネント、およびデバイス上の他のアプリケーションのログが含まれます。コンポーネントは Greengrass nucleus の一部ではありません。

詳細については、「コンポーネントタイプ」を参照してください。

- `region` - コアデバイスが使用する AWS リージョン。
- `componentName` - コンポーネントの名前。システムログの場合、この値は `System` です。

ログストリーム名

```
/date/thing/thingName
```

ログストリーム名は次の変数を使用します。

- `date` - 2020/12/15 など、ログの日付。ログマネージャーコンポーネントは `yyyy/MM/dd` 形式を使用します。
- `thingName` - コアデバイスの名前。

> **Note**
>
> モノの名前にコロン (:) が含まれている場合、ログマネージャーはコロンをプラス (+) に置き換えます。

2.0.x

ロググループ名

```
/aws/greengrass/componentType/region/componentName
```

ロググループ名は、次の変数を使用します。

- `componentType` - 次のいずれかに該当するコンポーネントのタイプ。
  - `GreengrassSystemComponent` - コンポーネントは Greengrass nucleus の一部です。
  - `UserComponent` - コンポーネントは Greengrass nucleus の一部ではありません。ログマネージャーは、デバイスの Greengrass コンポーネントと他のアプリケーションにこのタイプを使用します。
- `region` - コアデバイスが使用する AWS リージョン。

- componentName - コンポーネントの名前。システムログの場合、この値は System です。

ログストリーム名

```
/date/deploymentTargets/thingName
```

ログストリーム名は次の変数を使用します。

- date - 2020/12/15 など、ログの日付。ログマネージャーコンポーネントは yyyy/MM/dd 形式を使用します。
- deploymentTargets - デプロイにコンポーネントが含まれるモノ。ログマネージャーコンポーネントは、各ターゲットをスラッシュで区切ります。ローカルデプロイの結果としてコンポーネントがコアデバイスに実行される場合、この値は LOCAL_DEPLOYMENT です。

  MyGreengrassCore という名前のコアデバイスがあり、そのコアデバイスに 2 つのデプロイがある例を見てみます。

  - コアデバイス MyGreengrassCore をターゲットするデプロイ。
  - コアデバイスを含む MyGreengrassCoreGroup という名前のモノグループをターゲットするデプロイ。

  このコアデバイスの deploymentTargets は thing/MyGreengrassCore/thinggroup/MyGreengrassCoreGroup です。
- thingName - コアデバイスの名前。

ログエントリのフォーマット。

Greengrass nucleus は、文字列または JSON 形式でログファイルを書きます。システムログの場合、logging エントリーの format フィールドを設定することでフォーマットを制御します。Greengrass nucleus コンポーネントの設定ファイルに logging エントリが記載されています。詳細については、Greengrass nucleus 設定を参照してください。

テキスト形式は自由形式で、任意の文字列を受け付けます。次のフリート状況サービスメッセージは、文字列形式の口ギングの例です:

```
2023-03-26T18:18:27.271Z [INFO] (pool-1-thread-2)
com.aws.greengrass.status.FleetStatusService: fss-status-update-published.
Status update published to FSS. {trigger=CADENCE, serviceName=FleetStatusService,
currentState=RUNNING}
```

Greengrass CLI のログコマンドでログを表示したり、プログラムでログを操作する場合は、JSON 形式を使用する必要があります。次の例では、JSON の形状の概要を説明します：

```
{
   "loggerName": <string>,
   "level": <"DEBUG" | "INFO" | "ERROR" | "TRACE" | "WARN">,
   "eventType": <string, optional>,
   "cause": <string, optional>,
   "contexts": {},
   "thread": <string>,
   "message": <string>,
   "timestamp": <epoch time> # Needs to be epoch time
}
```

コンポーネントのログの出力を制御するために、minimumLogLevel 設定オプションを使用することができます。このオプションを使用するには、コンポーネントが JSON 形式でログエントリーを書き込む必要があります。システムログファイルと同じ形式を使用する必要があります。

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.3.7	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.3.6	バグ修正と機能向上 • 特定のエラーのログレベルを調整します。
2.3.5	改良点 ログのアップロード速度が向上します。 Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.3.4	バグ修正と機能向上 • `periodicUploadIntervalSec` パラメータを小数値に設定するためのサポートを追加します。最小値は 1 マイクロ秒です。 • ログマネージャーが`putLogEvents` 制限を尊重しない問題を修正しました CloudWatch。
2.3.3	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.3.2	バグ修正と機能向上 • ログファイルがアップロードされる前に削除されないように、容量管理を改善しました。 • キャッシュ管理に関する問題を修正しました。

バージョン	変更
	• マイナーなバグの追加修正と機能向上。
2.3.1	バグ修正と機能向上 • 複数のアクティブなログファイルを持つファイルグループをターゲットとする が、重複するエントリを にアップロードする問題を修正しました CloudWatch。 • マイナーなバグの追加修正と機能向上。
2.3.0	**Note** ログマネージャー 2.3.0 にアップグレードする場合は、Greengrass nucleus 2.9.1 にアップグレードすることをお勧めします。 新機能 新しいファイルがローテーションされるのを待つことなく、アクティブなログファイルを処理して直接アップロードすることによって、ログの遅延を軽減します。 バグ修正と機能向上 • 一意の名前を持つファイルをローテーションする際のログローテーションのサポートを改善しました。 • マイナーなバグの追加修正と機能向上。
2.2.8	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.2.7	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.2.6	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.2.5	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.2.4	バグ修正と機能向上 • 無効な設定を処理するときの安定性を向上させます。 • 追加のマイナー修正と機能向上。
2.2.3	バグ修正と機能向上 • コンポーネントが再起動またはエラーに直面するときに特定のシナリオで安定性を向上させます。 • 特定のシナリオで、大きなログメッセージと大きなログファイルがアップロードされない問題を修正します。 • このコンポーネントが設定リセット更新を処理する際に関する問題を修正しました。 • `null` `diskSpaceLimit` 設定値がコンポーネントのデプロイを妨げる問題を修正します。
2.2.2	バグ修正と機能向上 • 256 キロバイトを超えるログメッセージのサポートを追加します。ログマネージャーコンポーネントは、これらの大きなログメッセージを、同じログイベントのタイムスタンプを持つ複数のメッセージに分割します。
2.2.1	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.2.0	新機能 • `componentLogsConfigurationMap` 設定パラメータを追加して、コンポーネントのログ設定のマップ形式をサポートします。マップの各 `componentName` オブジェクトは、コンポーネントまたはアプリケーションのログ設定を定義します。
2.1.3	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.1.1	バグ修正と機能向上 • 特定のケースでシステムログ設定が更新されなかった問題を修正します。
2.1.0	バグ修正と機能向上 • 標準出力 (stdout) と標準エラー (stderr) で出力する Greengrass コンポーネントに機能する `logFileDirectoryPath` と `logFileRegex` にデフォルトを使用します。 • ログを CloudWatch Logs にアップロードするときに、設定済みのネットワークプロキシを介してトラフィックを正しくルーティングします。 • ログストリーム名でコロン文字 (:) を正しく処理します。 CloudWatch Logs ログストリーム名はコロンをサポートしていません。 • ログストリームからモノグループ名を削除して、ログストリーム名を簡素化します。 • 通常の動作中に出力されるエラーログメッセージを削除します。
2.0.x	当初のバージョン

機械学習コンポーネント

AWS IoT Greengrass は、Amazon でトレーニングされたモデル、 SageMaker または Amazon S3 に保存されている独自の事前トレーニング済みモデルを使用して機械学習推論を実行するために、サポートされているデバイスにデプロイできる次の機械学習コンポーネントを提供します。

AWS は、次の機械学習コンポーネントのカテゴリを示します。

- モデルコンポーネント - Greengrass アーティファクトとして機械学習モデルが含まれます。
- ランタイムコンポーネント - 機械学習フレームワークとその従属関係を Greengrass コアデバイスにインストールするスクリプトが含まれます。
- 推論コンポーネント - 推論コードが含まれ、機械学習フレームワークをインストールして事前学習済みの機械学習モデルをダウンロードするためのコンポーネント従属関係が含まれます。

AWSが提供する機械学習コンポーネントでサンプル推論コードと事前トレーニング済みモデルを使用して、DLR と TensorFlow Lite を使用してイメージ分類とオブジェクト検出を実行できます。Amazon S3 に格納されている独自のモデルでカスタム機械学習の推論を実行したり、別の機械学習フレームワークを使用したりするには、これらのパブリックコンポーネントのレシピをテンプレートとして使用して、カスタム機械学習コンポーネントを作成できます。詳細については、「機械学習コンポーネントのカスタマイズ」を参照してください。

AWS IoT Greengrass には、Greengrass コアデバイス上の SageMaker Edge Manager エージェントのインストールとライフサイクルを管理するための AWSが提供するコンポーネントも含まれています。 SageMaker Edge Manager では、Amazon SageMaker Neo でコンパイルされたモデルをコアデバイス上で直接使用できます。詳細については、「Greengrass コアデバイスで Amazon SageMaker Edge Manager を使用する」を参照してください。

次の表では、AWS IoT Greengrass で利用できる機械学習コンポーネントを一覧表示します。

> **Note**
>
> AWS が提供する複数のコンポーネントは、Greengrass nucleus の特定マイナーバージョンに依存します。この従属関係により、Greengrass nucleus を新しいマイナーバージョンに更新するとき、これらのコンポーネントを更新する必要があります。各コンポーネントが依存する nucleus の特定バージョンの情報については、対応するコンポーネントのトピックを参照してください。nucleus の更新の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネントのコンポーネントタイプが汎用と Lambda の両方である場合、コンポーネントの現在のバージョンは汎用タイプであり、コンポーネントの以前のバージョンは Lambda タイプです。

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
Lookout for Vision エッジエージェント	Amazon Lookout for Vision ランタイムを Greengrass コアデバイスにデプロイし、	ジェネリック	Linux	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
	コンピュータビジョンを使用して産業製品の欠陥を検出できるようにします。			
SageMaker Edge マネージャー	Amazon SageMaker Edge Manager エージェントを Greengrass コアデバイスにデプロイします。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
DLR イメージ分類	DLR イメージ分類モデルストアと DLR ランタイムコンポーネントを従属関係として使用し、DLR をインストール、サンプルイメージ分類モデルをダウンロード、サポートされているデバイスにイメージ分類推論を実行する推論コンポーネント。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
DLR オブジェクトの検出	DLR オブジェクト検知モデルストアと DLR ランタイムコンポーネントを従属関係として使用し、DLR をインストール、サンプルオブジェクト検知モデルをダウンロード、サポートされているデバイスにオブジェクト検知推論を実行する推論コンポーネント。	ジェネリック	Linux、Windows	いいえ
DLR イメージ分類モデルストア	Greengrass アーティファクトとしてサンプル ResNet-50 イメージ分類モデルを含むモデルコンポーネント。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
DLR オブジェクト検出モデルストア	Greengrass アーティファクトとしてサンプル YOLOv3 オブジェクト検出モデルを含むモデルコンポーネント。	ジェネリック	Linux、Windows	いいえ
DLR ランタイム	DLR とその従属関係を Greengrass コアデバイスにインストールするために使用されるインストールスクリプトを含むランタイムコンポーネント。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
TensorFlow Lite イメージ分類	TensorFlow Lite イメージ分類モデルストアと TensorFlow Lite ランタイムコンポーネントを依存関係として使用し、TensorFlow Lite をインストールし、サンプルイメージ分類モデルをダウンロードし、サポートされているデバイスにイメージ分類推論を実行する推論コンポーネント。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
TensorFlow ライトオブジェクト検出	TensorFlow Lite オブジェクト検出モデルストアと TensorFlow Lite ランタイムコンポーネントを依存関係として使用し、TensorFlow Lite をインストールし、サンプルオブジェクト検出モデルをダウンロードし、サポートされているデバイスでオブジェクト検出推論を実行する推論コンポーネント。	ジェネリック	Linux、Windows	いいえ
TensorFlow Lite イメージ分類モデルストア	Greengrass アーティファクトとしてサンプル MobileNet v1 モデルを含むモデルコンポーネント。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
TensorFlow Lite オブジェクト検出モデルストア	Greengrass アーティファクトとしてサンプルのシングルショット検出 (SSD) MobileNet モデルを含むモデルコンポーネント。	ジェネリック	Linux、Windows	いいえ
TensorFlow Lite ランタイム	Greengrass コアデバイスに TensorFlow Lite とその依存関係をインストールするために使用されるインストールスクリプトを含むランタイムコンポーネント。	ジェネリック	Linux、Windows	いいえ

Lookout for Vision エッジエージェント

Lookout for Vision エッジエージェントのコンポーネント (`aws.iot.lookoutvision.EdgeAgent`) は、ローカルの Amazon Lookout for Vision ランタイムサーバーをインストールし、コンピュータビジョンを使用して産業製品の視覚的な欠陥を検出します。

このコンポーネントを使用するには、Lookout for Vision の機械学習モデルコンポーネントを作成してデプロイします。これらの機械学習モデルは、モデルのトレーニングに使用する画像のパターンを見つけることによって、画像の異常の存在を予測します。次に、クライアント アプリケーション コンポーネントと呼ばれるカスタム Greengrass コンポーネントを開発してデプロイできます。このカ

スタム Greengrass コンポーネントは、このランタイムコンポーネントにイメージとビデオストリームを提供して、機械学習モデルを使用して異常を検出します。

Lookout for Vision エッジエージェント API を使用して、他の Greengrass コンポーネントからこのコンポーネントとやり取りできます。この API は gRPC を使用して実装され、これは遠隔でプロシージャ呼び出しを行うためのプロトコルです。詳細については、「Amazon Lookout for Vision デベロッパーガイド」の「クライアントアプリケーションコンポーネントの記述」と「Lookout for Vision エッジエージェント API リファレンス」を参照してください。

このコンポーネントの使用方法の詳細については、次の内容を参照してください:

- Greengrass コアデバイスのAmazon Lookout for Vision を使用
- 「Amazon Lookout for Vision デベロッパーガイド」の「Amazon Lookout for Vision とは」
- 「Amazon Lookout for Vision デベロッパーガイド」の「Lookout for Vision モデルの作成」。
- 「Amazon Lookout for Vision デベロッパーガイド」の「エッジデバイスで Lookout for Vision モデルの使用」。

> **Note**
>
> Lookout for Vision Edge Agent コンポーネントは、次の でのみ使用できます AWS リージョン。
>
> - 米国東部 (オハイオ)
> - 米国東部 (バージニア北部)
> - 米国西部 (オレゴン)
> - 欧州 (フランクフルト)
> - 欧州 (アイルランド)
> - アジアパシフィック (東京)
> - アジアパシフィック (ソウル)

トピック

- バージョン
- タイプ
- オペレーティングシステム

- 要件
- 依存関係
- 構成
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 1.2.x
- 1.1.x
- 1.0.x
- 0.1.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、Linux コアデバイスにのみインストールできます。

要件

このコンポーネントには次の要件があります。

- Greengrass コアデバイスは、Armv8 (AArch64) または x86_64 アーキテクチャを使用する必要があります。
- このコンポーネントのバージョン 1.0.0 以降を使用する場合、Greengrass コアデバイスにインストールされた Python 3.8 または Python 3.9 (`pip` を含む)。

  このコンポーネントのバージョン 0.1.x を使用する場合、Greengrass コアデバイスにインストールされた Python 3.7 (`pip` を含む)。

> **Important**
>
> デバイスには、これらに一致した Python のバージョンがいずれか必要です。このコンポーネントは、Python の新しいバージョンをサポートしていません。

- グラフィックスプロセッシングユニット (GPU) 推論を使用するには、コアデバイスが次の要件を満たしている必要があります。GPU 推論は、このコンポーネントのバージョン 1.1.0 以降でオプションとして使用できます。
  - CUDA をサポートするグラフィックス処理ユニット (GPU)。詳細については、「CUDA Toolkit マニュアル」の「CUDA 対応の GPU を持っていることを確認」を参照してください。
  - Greengrass コアデバイスにインストールされた cuDNN、CUDA、TensorRT。
    - Jetson Nano や Jetson Xavier、cuDNN、CUDA、TensorRT などの NVIDIA Jetson デバイスには、NVIDIA が付属しています JetPack。変更を一切する必要はありません。このコンポーネントは JetPack 4.4、JetPack4.5、JetPack 4.5.1、および JetPack4.6.1 をサポートしています。

      > **Important**
      >
      > 別のバージョンではなく JetPack 、これらのバージョンの 1 つをインストールする必要があります。Lookout for Vision サービスは、これらの JetPackプラットフォーム用のコンピュータビジョンモデルをコンパイルします。

    - NVIDIA アンペアマイクロアーキテクチャ (または GPU のコンピューティング能力が 8.0 に等しい場合) を備えた GPU 付き x86 デバイスに対して、次の手順を実行します。
      - 「NVIDIA cuDNN Installation Guide」の指示に従って、cuDNN をインストールします。
      - 「Linux 用 NVIDIA CUDA インストールガイド」の指示に従って、CUDA バージョン 11.2 をインストールします。
      - 「NVIDIA TensorRT マニュアル」の指示に従って、TensorRT バージョン 8.2.0 をインストールします。
    - Ampere (または GPU のコンピューティング能力が 8.0未満の場合) 以前の NVIDIA マイクロアーキテクチャを備えた GPU 付き x86 デバイスに対して、次の手順を実行します:
      - 「NVIDIA cuDNN Installation Guide」の指示に従って、cuDNN をインストールします。
      - 「Linux 用 NVIDIA CUDA インストールガイド」の指示に従って、CUDA バージョン 10.2 をインストールします。

  - 「NVIDIA TensorRT マニュアル」の指示に従って、TensorRT バージョン 8.0.0 より前のバージョン 7.1.3 以降をインストールします。
- このコンポーネントを実行するシステムユーザーは、デバイスの GPU にアクセスできるシステムグループのメンバーである必要があります。このグループの名前は、オペレーティングシステムによって異なります。オペレーティングシステムと GPU のマニュアルを参照して、このシステムグループの名前を確認してください。

  例えば、NVIDIA Jetson デバイスにおいて、このグループの名前は `video` であり、次のコマンドを実行してシステムユーザをこのグループに追加できます。*`ggc_user`* を追加するユーザーの名前に置き換えます。

  ```
  sudo usermod -aG video ggc_user
  ```

依存関係

このコンポーネントに依存関係はありません。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

`Socket`

(オプション) エッジエージェントが動作するファイルソケット。Lookout for Vision モデルコンポーネントは、このファイルソケットを使用してエッジエージェントと通信します。このパラメータを変更する場合、Lookout for Vision モデルコンポーネントをデプロイするときに同じ値を指定する必要があります。

デフォルト: `unix:///tmp/aws.iot.lookoutvision.EdgeAgent.sock`

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

```
/greengrass/v2/logs/aws.iot.lookoutvision.EdgeAgent.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。を AWS IoT Greengrass ルートフォルダへのパス*/greengrass/v2*に置き換えます。

```
sudo tail -f /greengrass/v2/logs/aws.iot.lookoutvision.EdgeAgent.log
```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
1.2.0	一般的なバグ修正と機能強化。
1.1.9	一般的なバグ修正と機能強化。
1.1.8	一般的なバグ修正と機能強化。
1.1.7	新機能 • Lookout for Vision Edge Agent 仮想環境に opencv-python-headless パッケージをインストールします。 バグ修正と機能向上 • 信頼スコアの計算を改善しました。 • ヒートマップモデルマスクを元のファイルサイズに変更します。 • 一般的なバグ修正と機能強化。
1.1.6	新機能 DetectAnomalies の結果に新しい値を追加しました。 • anomaly_score — 画像がどの程度異常であるかを示す 0.0 ~ 1.0 の間の数値。 • anomaly_threshold — モデルトレーニング中に設定される、異常画像と正常画像の境界を決定するしきい値。

バージョン	変更
	一般的なバグ修正と機能強化。
1.1.4	新機能 OpenCV のサポートが追加されました (利用可能な場合)。エッジエージェントは OpenCV が利用できない場合に Pillow を使用します。 バグ修正と機能向上 一般的なバグ修正と機能強化。
1.1.3	一般的なバグ修正と機能強化。
1.1.1	一般的なバグ修正と機能強化。
1.1.0	新機能 • 画像セグメントモデルのサポートを追加。画像の異常を特定します。 • CPU 推論のサポートを追加。これにより、GPU のないコアデバイスで Lookout for Vision モデルを使用できます。 バグ修正と機能向上 • 一般的なバグ修正と機能強化。
1.0.0	Lookout for Vision エッジエージェントコンポーネントのこのバージョンは、バージョン 0.1.x とは異なる Python のバージョンが必要です。v0.1.x から v1.x にアップグレードする場合、コアデバイスにインストールされた Python をアップグレードする必要があります。 バグ修正と機能向上 • 一般的なバグ修正と機能強化。
0.1.37	一般的なバグ修正と機能強化。
0.1.36	当初のバージョン

SageMaker Edge マネージャー

> **Important**
>
> SageMaker Edge Manager は 2024 年 4 月 26 日に廃止されます。エッジデバイスにモデルをデプロイし続ける方法の詳細については、SageMaker 「 Edge Manager のサポート終了」を参照してください。

Amazon SageMaker Edge Manager コンポーネント (`aws.greengrass.SageMakerEdgeManager`) は、 SageMaker Edge Manager エージェントバイナリをインストールします。

SageMaker Edge Manager はエッジデバイスのモデル管理を提供するため、エッジデバイスのフリートで機械学習モデルを最適化、保護、モニタリング、維持できます。 SageMaker Edge Manager コンポーネントは、コアデバイスに SageMaker Edge Manager エージェントのライフサイクルをインストールおよび管理します。 SageMaker Edge Manager を使用して、 SageMaker Neo でコンパイルされたモデルを Greengrass コアデバイスのモデルコンポーネントとしてパッケージ化して使用することもできます。コアデバイスで SageMaker Edge Manager エージェントを使用する方法の詳細については、「」を参照してくださいGreengrass コアデバイスで Amazon SageMaker Edge Manager を使用する。

SageMaker Edge Manager コンポーネント v1.3.x は、Edge Manager エージェントバイナリ v1.20220822.836f3023 をインストールします。Edge Manager エージェントのバイナリバージョンの詳細については、「Edge Manager エージェント」を参照してください。

> **Note**
>
> SageMaker Edge Manager コンポーネントは、次の でのみ使用できますAWS リージョン。
>
> - 米国東部 (オハイオ)
> - 米国東部 (バージニア北部)
> - 米国西部 (オレゴン)
> - 欧州 (フランクフルト)
> - 欧州 (アイルランド)
> - アジアパシフィック (東京)

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 1.3.x
- 1.2.x
- 1.1.x
- 1.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Amazon Linux 2 で実行されている Greengrass コアデバイス、Debian ベースの Linux プラットフォーム (x86_64 またはArmv8)、または Windows (x86_64)。アカウントをお持ちでない場合は、「チュートリアル: AWS IoT Greengrass V2 の開始方法」を参照してください。
- Python 3.6 以降 (ご使用の Python のバージョン用 `pip` がコアデバイスにインストールされていること)。
- 次のように設定された Greengrass デバイスのロール:
  - 次の IAM ポリシーの例で示されているように、`credentials.iot.amazonaws.com` と `sagemaker.amazonaws.com` がロールの継承を可能にする信頼関係。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.iot.amazonaws.com"
       },
      "Action": "sts:AssumeRole"
    },
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "sagemaker.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

  - IAM AmazonSageMakerEdgeDeviceFleetPolicy マネージドポリシー。
  - 次の IAM ポリシーの例で示されている `s3:PutObject` アクション。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:PutObject"
      ],
      "Resource": [
        "*"
```

```
      ],
      "Effect": "Allow"
    }
  ]
}
```

- Greengrass コアデバイスAWS リージョンと同じ AWS アカウントと で作成された Amazon S3 バケット。 SageMaker Edge Manager では、エッジデバイスフリートを作成し、デバイスで実行中の推論からサンプルデータを保存するために S3 バケットが必要です。S3 バケットを作成する方法の情報については、「Amazon S3 の使用を開始」を参照してください。
- Greengrass コアデバイスと同じAWS IoTロールエイリアスを使用する SageMaker エッジデバイスフリート。詳細については、「エッジデバイスフリートを作成する」を参照してください。
- Greengrass コアデバイスは、 SageMaker エッジデバイスフリートにエッジデバイスとして登録されています。エッジデバイス名は、コアデバイスの AWS IoT モノ名に一致する必要があります。詳細については、「Greengrass コアデバイスを登録する」を参照してください。

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
`edge.sagemaker.`*`region`*`.amazonaws.com`	443	はい	デバイス登録ステータスを確認し、 にメトリクスを送信します SageMaker 。
`*.s3.amazonaws.com`	443	はい	指定した S3 バケットにキャプチャデータをアップロ

エンドポイント	ポート	必要	説明
			ードします。 * は、データをアップロードする各バケットの名前に置き換えることができます。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

1.3.5

次の表に、このコンポーネントのバージョン 1.3.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト
トークン交換サービス	>=0.0.0	ハード

1.3.4

次の表に、このコンポーネントのバージョン 1.3.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト
トークン交換サービス	>=0.0.0	ハード

1.3.3

次の表に、このコンポーネントのバージョン 1.3.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト
トークン交換サービス	>=0.0.0	ハード

1.3.2

次の表に、このコンポーネントのバージョン 1.3.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト
トークン交換サービス	>=0.0.0	ハード

1.3.1

次の表に、このコンポーネントのバージョン 1.3.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト
トークン交換サービス	>=0.0.0	ハード

1.1.1 - 1.3.0

次の表に、このコンポーネントのバージョン 1.1.1 ~ 1.3.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト
トークン交換サービス	>=0.0.0	ハード

1.1.0

次の表に、このコンポーネントのバージョン 1.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト
トークン交換サービス	>=0.0.0	ハード

1.0.3

次の表に、このコンポーネントのバージョン 1.0.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト
トークン交換サービス	>=0.0.0	ハード

1.0.1 and 1.0.2

次の表に、このコンポーネントのバージョン 1.0.1 および 1.0.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト

依存関係	互換性のあるバージョン	依存関係タイプ
トークン交換サービス	>=0.0.0	ハード

1.0.0

次の表に、このコンポーネントのバージョン 1.0.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト
トークン交換サービス	>=0.0.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

> **Note**
>
> このセクションでは、コンポーネントで設定する設定パラメータについて説明します。対応する SageMaker Edge Manager 設定の詳細については、「Amazon SageMaker デベロッパーガイド」の「Edge Manager エージェント」を参照してください。

`DeviceFleetName`

Greengrass コアデバイスを含む SageMaker Edge Manager デバイスフリートの名前。

このコンポーネントをデプロイするときは、 設定の更新でこのパラメータの値を指定する必要があります。

`BucketName`

キャプチャした推論データのアップロード先の S3 バケットの名前。バケット名には文字列 `sagemaker` が含まれている必要があります。

CaptureDataDestination を Cloud に設定した場合、または CaptureDataPeriodicUpload を true に設定した場合、このコンポーネントをデプロイするときに、設定の更新でこのパラメータの値を指定する必要があります。

> **Note**
>
> キャプチャデータは、今後の分析のために S3 バケットまたはローカルディレクトリに推論入力、推論結果、および追加の推論データをアップロードするために使用する SageMaker の機能です。 SageMaker Edge Manager でキャプチャデータを使用する方法の詳細については、「Amazon SageMaker デベロッパーガイド」の「モデルの管理」を参照してください。

CaptureDataBatchSize

(オプション) エージェントが処理するキャプチャデータ要求のバッチのサイズ。この値は、CaptureDataBufferSize で指定したバッファサイズよりも小さくする必要があります。バッファサイズの半分を超えないことをお勧めします。

エージェントは、バッファ内の要求数が CaptureDataBatchSize の数に達したとき、または CaptureDataPushPeriodSeconds の間隔が経過したとき、いずれか早いほうの要求バッチを処理します。

デフォルト: 10

CaptureDataBufferSize

(オプション) バッファに保存されているキャプチャデータ要求の最大数。

デフォルト: 30

CaptureDataDestination

(オプション) キャプチャしたデータを保存する保存先。このパラメータには以下の値があります。

- Cloud - BucketName で指定した S3 バケットにキャプチャデータをアップロードします。
- Disk - キャプチャしたデータをコンポーネントの作業ディレクトリに書き込みます。

Disk を指定した場合、CaptureDataPeriodicUpload を true に設定して、キャプチャしたデータを S3 バケットに定期的にアップロードすることも選択できます。

デフォルト: Cloud

`CaptureDataPeriodicUpload`

(任意) キャプチャされたデータを定期的にアップロードするかどうかを指定する文字列値。サポートされている値は、`true` および `false` です。

`CaptureDataDestination` を `Disk` に設定し、エージェントがキャプチャしたデータを S3 バケットに定期的にアップロードする場合は、このパラメータを `true` に設定します。

デフォルト: `false`

`CaptureDataPeriodicUploadPeriodSeconds`

(オプション) SageMaker Edge Manager エージェントがキャプチャしたデータを S3 バケットにアップロードする秒単位の間隔。`CaptureDataPeriodicUpload` を `true` に設定する場合は、このパラメータを使用します。

デフォルト: 8

`CaptureDataPushPeriodSeconds`

(オプション) SageMaker Edge Manager エージェントがバッファからのキャプチャデータリクエストのバッチを処理する秒単位の間隔。

エージェントは、バッファ内の要求数が `CaptureDataBatchSize` の数に達したとき、または `CaptureDataPushPeriodSeconds` の間隔が経過したとき、いずれか早いほうの要求バッチを処理します。

デフォルト: 4

`CaptureDataBase64EmbedLimit`

(オプション) SageMaker Edge Manager エージェントがアップロードするキャプチャデータの最大バイトサイズ。

デフォルト: 3072

`FolderPrefix`

(オプション) エージェントがキャプチャしたデータを書き込むフォルダの名前。`CaptureDataDestination` を `Disk` に設定すると、エージェントは `CaptureDataDiskPath` で指定されたディレクトリにフォルダを作成します。`CaptureDataDestination` を `Cloud` に設定した場合、または `CaptureDataPeriodicUpload` を `true` に設定した場合、エージェントはS3バケットにフォルダを作成します。

デフォルト: `sme-capture`

`CaptureDataDiskPath`

この機能は、 SageMaker Edge Manager コンポーネントの v1.1.0 以降のバージョンで使用できます。

(オプション) エージェントがキャプチャされたデータフォルダを作成するフォルダへのパス。`CaptureDataDestination` を `Disk` に設定すると、エージェントはこのディレクトリにキャプチャされたデータフォルダを作成します。この値を指定しない場合、エージェントがキャプチャデータフォルダをコンポーネントの作業ディレクトリに作成します。`FolderPrefix` パラメータを使用して、キャプチャされたデータフォルダの名前を指定します。

デフォルト: */greengrass/v2*`/work/aws.greengrass.SageMakerEdgeManager/capture`

`LocalDataRootPath`

この機能は、 SageMaker Edge Manager コンポーネントの v1.2.0 以降のバージョンで使用できます。

(オプション) このコンポーネントがコアデバイス上の次のデータを保存するパス。

- `DbEnable` を `true` に設定したときのランタイムデータのローカルデータベース。
- SageMaker `DeploymentEnable`を に設定すると、このコンポーネントが自動的にダウンロードする Neo コンパイル済みモデル`true`。

デフォルト: */greengrass/v2*`/work/aws.greengrass.SageMakerEdgeManager`

`DbEnable`

(オプション) このコンポーネントを有効にすると、コンポーネントに障害が発生したり、デバイスの電源が切れたりした場合に備えて、ランタイムデータをローカルデータベースに保存してデータを保全できます。

このデータベースには、コアデバイスのファイルシステムに 5 MB のストレージが必要です。

デフォルト: `false`

`DeploymentEnable`

この機能は、 SageMaker Edge Manager コンポーネントの v1.2.0 以降のバージョンで使用できます。

(オプション) このコンポーネントを有効にして、Amazon S3 にアップロードした から SageMaker Neo でコンパイルされたモデルを自動的に取得できます。新しいモデルを Amazon

S3 にアップロードしたら、Studio または SageMaker API を使用して SageMaker、新しいモデルをこのコアデバイスにデプロイします。この機能を有効にすると、AWS IoT Greengrass デプロイを作成する必要なく、新しいモデルをコアデバイスにデプロイできます。

> **Important**
>
> この機能を使用するには、`DbEnable` を `true` に設定する必要があります。この機能では、ローカルデータベースを使用して、AWS クラウド から取得したモデルを追跡します。

デフォルト: `false`

`DeploymentPollInterval`

この機能は、 SageMaker Edge Manager コンポーネントの v1.2.0 以降のバージョンで使用できます。

(オプション) このコンポーネントがダウンロードする新しいモデルをチェックする間隔 (分単位)。このオプションは、`DeploymentEnable` を `true` に設定すると適用されます。

デフォルト: `1440` (1 日)

`DLRBackendOptions`

この機能は、 SageMaker Edge Manager コンポーネントの v1.2.0 以降のバージョンで使用できます。

(オプション) このコンポーネントが使用する DLR ランタイムに設定する DLR ランタイムフラグ。次のフラグを設定できます。

- TVM_TENSORRT_CACHE_DIR – TensorRT モデルのキャッシュを有効にします。読み取り/書き込みアクセス許可を持つ既存のフォルダへの絶対パスを指定します。
- TVM_TENSORRT_CACHE_DISK_SIZE_MB – TensorRT モデルのキャッシュフォルダの上限を割り当てます。ディレクトリサイズがこの制限を超えると、最も使用頻度の低いキャッシュされたエンジンが削除されます。デフォルト値は 512 MB です。

例えば、このパラメータを次の値に設定すると、TensorRT モデルのキャッシュを有効にし、キャッシュサイズを 800 MB に制限できます。

```
TVM_TENSORRT_CACHE_DIR=/data/secured_folder/trt/cache;
 TVM_TENSORRT_CACHE_DISK_SIZE_MB=800
```

`SagemakerEdgeLogVerbose`

(オプション) デバッグログ記録を有効にするかどうかを指定する文字列値。サポートされている値は、`true` および `false` です。

デフォルト: `false`

`UnixSocketName`

(オプション) コアデバイス上の SageMaker Edge Manager ソケットファイル記述子の場所。

デフォルト: `/tmp/aws.greengrass.SageMakerEdgeManager.sock`

Example 例: 設定マージの更新

次の設定例では、コアデバイスが の一部であり、エージェントがキャプチャデータをデバイス*MyEdgeDeviceFleet*と S3 バケットの両方に書き込むことを指定します。この設定により、デバッグログ記録も有効になります。

```
{
    "DeviceFleetName": "MyEdgeDeviceFleet",
    "BucketName": "DOC-EXAMPLE-BUCKET",
    "CaptureDataDestination": "Disk",
    "CaptureDataPeriodicUpload": "true",
    "SagemakerEdgeLogVerbose": "true"
}
```

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/aws.greengrass.SageMakerEdgeManager.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.SageMakerEdgeManager.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.SageMakerEdgeManager.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.greengrass.SageMakerEdgeManager.log -Tail
   10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
1.3.5	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
1.3.4	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
1.3.3	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
1.3.2	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
1.3.1	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
1.3.0	新機能 • TensorRT のキャッシュディスクサイズ管理のサポートを追加します。

バージョン	変更
	• DLR BackendOptions パラメータにオプションの TVM_TENSORRT_CACHE_DISK_SIZE_MB フラグを追加して、ディスクにキャッシュされるモデルのサイズ制限を設定します。 改良点 • 予測の同時実行を改善します。これにより、GPU などのデバイスアクセラレータエンジンをより有効に活用できるようになります。
1.2.0	新機能 • Amazon S3 にアップロードする SageMaker Neo コンパイル済みモデルを自動的に取得するための、このコンポーネントのサポートが追加されました。この機能を有効にすると、AWS IoT Greengrass デプロイを作成する必要なく、新しいモデルをコアデバイスにデプロイできます。 • コンポーネントに障害が発生したり、デバイスの電源が切れたりした場合に、このコンポーネントがランタイムデータを保存するために使用するバックアップデータベースのサポートが追加されています。 • このコンポーネントを設定するとき、DLR ランタイムフラグを設定するためのサポートが追加されました。
1.1.1	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
1.1.0	新機能 • Amazon Linux 2 を実行している Greengrass コアデバイスのサポートを追加します。 • 新しい CaptureDataDiskPath 設定パラメータが追加されました。このパラメータを使用して、デバイスのキャプチャデータフォルダのパスを指定できます。 バグ修正と機能向上 • Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
1.0.3	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。

バージョン	変更
1.0.2	バグ修正と機能向上 コンポーネント ライフサイクルのインストールスクリプトを更新します。コアデバイスには Python 3.6 以降をインストールする必要があり、このコンポーネントをデプロイする前に、デバイスにはお使いの Python バージョン向けの `pip` がインストールされている必要があります。
1.0.1	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
1.0.0	当初のバージョン

DLR イメージ分類

DLR イメージ分類コンポーネント (`aws.greengrass.DLRImageClassification`) には、深層学習ランタイムと resnet-50 モデルを使ってイメージ分類の推論を行うためのサンプル推論コードが含まれています。このコンポーネントは、バリアント DLR イメージ分類モデルストア と DLR ランタイム コンポーネントを依存関係として使用することで、DLR とサンプルモデルをダウンロードします。

カスタムトレーニングされた DLR モデルでこの推論コンポーネントを使用するには、依存モデルストアコンポーネントのカスタムバージョンを作成します。独自のカスタム推論コードを使用するには、このコンポーネントの recipe をテンプレートとして使用して、[create a custom inference component] (カスタム推論コンポーネントを作成) できます。

トピック

- バージョン
- タイプ
- オペレーティングシステム
- 要件
- 依存関係
- 構成
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x
- 2.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Amazon Linux 2 または Ubuntu 18.04 を実行している Greengrass コアデバイスの場合は、GNU C ライブラリ (glibc) バージョン 2.27 以降がデバイスにインストールされている必要があります。
- Raspberry Pi などの Armv7l デバイスでは、OpenCV-Python の依存関係がデバイスにインストールされています。次のコマンドを実行して依存関係をインストールします。

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

- Raspberry Pi OS Bullseye を実行する Raspberry Pi デバイスでは、次の要件を満たす必要があります。
  - NumPy 1.22.4 以降がデバイスにインストールされている。Raspberry Pi OS Bullseye には以前のバージョンの が含まれているため NumPy、次のコマンドを実行して NumPy デバイスでアップグレードできます。

```
pip3 install --upgrade numpy
```

- デバイスで、レガシーカメラスタックが有効になっていること。Raspberry Pi OS Bullseye には、デフォルトで新しいカメラスタックが含まれており有効化されていますが、これには互換性がないため、レガシーカメラスタックを有効にしておく必要があります。

  レガシーカメラスタックを有効にするには

  1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

     ```
     sudo raspi-config
     ```

  2. [Interface Options] (インターフェイスオプション) を選択します。
  3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
  4. Raspberry Pi を再起動します。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.13

次の表に、このコンポーネントのバージョン 2.1.13 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト
DLR イメージ分類モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.12

次の表に、このコンポーネントのバージョン 2.1.12 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト
DLR イメージ分類モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.11

次の表に、このコンポーネントのバージョン 2.1.11 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト
DLR イメージ分類モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.10

次の表に、このコンポーネントのバージョン 2.1.10 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト
DLR イメージ分類モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.9

次の表に、このコンポーネントのバージョン 2.1.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト
DLR イメージ分類モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.8

次の表に、このコンポーネントのバージョン 2.1.8 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト
DLR イメージ分類モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト
DLR イメージ分類モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.6

次の表に、このコンポーネントのバージョン 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト
DLR イメージ分類モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.4 - 2.1.5

次の表に、このコンポーネントのバージョン 2.1.4 から 2.1.5 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト
DLR イメージ分類モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト
DLR イメージ分類モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト
DLR イメージ分類モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト
DLR イメージ分類モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.0.x

次の表に、このコンポーネントのバージョン 2.0.x の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	~2.0.0	ソフト
DLR イメージ分類モデルストア	~2.0.0	ハード
DLR	~1.3.0	ソフト

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

2.1.x

`accessControl`

(オプション) コンポーネントがデフォルトの通知トピックにメッセージをパブリッシュできるようにする承認ポリシーを含むオブジェクト。

デフォルト:

```
{
   "aws.greengrass.ipc.mqttproxy": {
      "aws.greengrass.DLRImageClassification:mqttproxy:1": {
         "policyDescription": "Allows access to publish via topic ml/dlr/image-
classification.",
         "operations": [
            "aws.greengrass#PublishToIoTCore"
         ],
         "resources": [
            "ml/dlr/image-classification"
         ]
      }
   }
}
```

`PublishResultsOnTopic`

(オプション) 推論結果をパブリッシュするトピック。この値を変更する場合は、カスタムトピック名と一致するように `resources` パラメータの `accessControl` の値も変更する必要があります。

デフォルト: `ml/dlr/image-classification`

`Accelerator`

使用するアクセラレーター。サポートされている値は、`cpu` および `gpu` です。

依存モデルコンポーネントのサンプルモデルは CPU アクセラレーションのみをサポートします。別のカスタムモデルで GPU アクセラレーションを使用するには、カスタムモデルコンポーネントを作成して、パブリックモデルコンポーネントを上書きします。

デフォルト: cpu

`ImageDirectory`

(オプション) 推論コンポーネントがイメージを読み取る場所と成る、デバイス上のフォルダへのパス。この値は、読み取り/書き込みアクセス権を持つデバイスの任意の場所に変更できます。

デフォルト: */greengrass/v2*/packages/artifacts-unarchived/*component-name*/image_classification/sample_images/

> **Note**
>
> `UseCamera` を `true` の値に設定した場合、この設定パラメータは無視されます。

`ImageName`

(オプション) 推論コンポーネントが予測を行う際の入力として使用するイメージの名前。コンポーネントは、`ImageDirectory` で指定されたフォルダ内のイメージを検索します。デフォルトでは、コンポーネントはデフォルトのイメージディレクトリにあるサンプルイメージを使用します。AWS IoT Greengrass は `jpeg`、`jpg`、`png`、および `npy` のイメージ形式をサポートします。

デフォルト: `cat.jpeg`

> **Note**
>
> `UseCamera` を `true` の値に設定した場合、この設定パラメータは無視されます。

`InferenceInterval`

(オプション) 推論コードによって行われた各予測間の時間 (秒単位)。サンプル推論コードは無期限に実行され、指定された時間間隔で予測を繰り返します。例えば、カメラで撮影したイメージをリアルタイム予測に使用する場合などには、この間隔を短い間隔に変更できます。

デフォルト: `3600`

`ModelResourceKey`

(オプション) 依存パブリックモデルコンポーネントで使用されるモデル。このパラメータを変更するのは、パブリックモデルコンポーネントをカスタムコンポーネントでオーバーライドする場合のみです。

デフォルト:

```
{
    "armv7l": "DLR-resnet50-armv7l-cpu-ImageClassification",
    "aarch64": "DLR-resnet50-aarch64-cpu-ImageClassification",
    "x86_64": "DLR-resnet50-x86_64-cpu-ImageClassification",
    "windows": "DLR-resnet50-win-cpu-ImageClassification"
}
```

`UseCamera`

(オプション) Greengrass コアデバイスに接続されたカメラの画像を使用するかどうかを定義する文字列値。サポートされている値は、`true` および `false` です。

この値を `true` に設定した場合、サンプル推論コードはデバイスのカメラにアクセスし、キャプチャしたイメージでローカルに推論を実行します。`ImageName` と `ImageDirectory` パラメータの値は無視されます。このコンポーネントを実行しているユーザーが、カメラがキャプチャしたイメージを保存する場所への、読み込み/書き込みアクセス権を持っていることを確認します。

デフォルト: `false`

> **Note**
>
> このコンポーネントの recipe を表示すると、`UseCamera` 設定パラメータはデフォルト設定には表示されません。ただし、このパラメータの値は、コンポーネントをデプロイするときに、[configuration merge update] (設定マージの更新) で変更することができます。
>
> `UseCamera` を `true` に設定する場合は、ランタイムコンポーネントによって作成された仮想環境から推論コンポーネントがカメラにアクセスできるようにするためのシンボルリンクも作成する必要があります。サンプル推論コンポーネントを使用したカメラの使用方法の詳細については、「コンポーネント設定の更新」を参照してください。

2.0.x

`MLRootPath`

(オプション) 推論コンポーネントがイメージを読み取り、推論結果を書き込む Linux コアデバイスのフォルダのパス。この値は、このコンポーネントを実行しているユーザーが読み取り/書き込みアクセス権を持つデバイスの任意の場所に変更できます。

デフォルト: *`/greengrass/v2`*`/work/variant.DLR/greengrass_ml`

デフォルト: *`/greengrass/v2`*`/work/variant.TensorFlowLite/greengrass_ml`

`Accelerator`

使用するアクセラレーター。サポートされている値は、cpu および gpu です。

依存モデルコンポーネントのサンプルモデルは CPU アクセラレーションのみをサポートします。別のカスタムモデルで GPU アクセラレーションを使用するには、カスタムモデルコンポーネントを作成して、パブリックモデルコンポーネントを上書きします。

デフォルト: cpu

`ImageName`

(オプション) 推論コンポーネントが予測を行う際の入力として使用するイメージの名前。コンポーネントは、`ImageDirectory` で指定されたフォルダ内のイメージを検索します。デフォルトの場所は *`MLRootPath`*`/images` です。AWS IoT Greengrass は `jpeg`、`jpg`、`png`、`npy` イメージ形式をサポートしています。

デフォルト: `cat.jpeg`

`InferenceInterval`

(オプション) 推論コードによって行われた各予測間の時間 (秒単位)。サンプル推論コードは無期限に実行され、指定された時間間隔で予測を繰り返します。例えば、カメラで撮影したイメージをリアルタイム予測に使用する場合などには、この間隔を短い間隔に変更できます。

デフォルト: `3600`

`ModelResourceKey`

(オプション) 依存パブリックモデルコンポーネントで使用されるモデル。このパラメータを変更するのは、パブリックモデルコンポーネントをカスタムコンポーネントでオーバーライドする場合のみです。

デフォルト:

```
armv7l: "DLR-resnet50-armv7l-cpu-ImageClassification"
x86_64: "DLR-resnet50-x86_64-cpu-ImageClassification"
```

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/aws.greengrass.DLRImageClassification.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.DLRImageClassification.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.DLRImageClassification.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.greengrass.DLRImageClassification.log -
  Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.13	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.1.12	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.11	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.10	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.9	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.8	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.1.7	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.6	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.1.5	コンポーネントはすべての AWS リージョン にリリースされています。
2.1.4	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。 このバージョンは欧州 (ロンドン) では利用できません (`eu-west-2`)。
2.1.3	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.1.1	新機能 • 深層学習ランタイム のバージョン 1.6.0.を使用します。 • Armv8 (AArch64) プラットフォームでのサンプルイメージ分類のサポートが追加されました。これにより、Jetson Nano などの NVIDIA Jetson を実行している Greengrass コアデバイスにまで機械学習サポートが拡張されます。 • サンプル推論でカメラ統合が可能になりました。新しい `UseCamera` 設定パラメータを使用すると、サンプル推論コードが Greengrass コアデバイスのカメラにアクセスできるようになり、キャプチャしたイメージ上でローカルに推論を実行できるようになります。 • AWS クラウド に推論結果をパブリッシュするためのサポートが追加されました。新しい `PublishResultsOnTopic` 設定パラメータを使用して、結果をパブリッシュするトピックを指定します。 • 推論を実行するイメージ用のカスタムディレクトリを指定できる新しい `ImageDirectory` 設定パラメータが追加されました。 バグ修正と機能向上 • 別の推論ファイルではなく、コンポーネントログファイルに推論結果を書き込みます。 • AWS IoT Greengrass Core ソフトウェアのログモジュールを使用してコンポーネントの出力を記録します。 • AWS IoT Device SDK を使用してコンポーネント設定を読み込み、設定への変更を適用します。
2.0.4	当初のバージョン

DLR オブジェクトの検出

DLR オブジェクト検出コンポーネント (`aws.greengrass.DLRObjectDetection`) には、深層学習ランタイムとサンプルの事前学習済みモデルを使用してオブジェクト検出推論を実行するサンプル推論コードが含まれています。このコンポーネントは、バリアント DLR オブジェクト検出モデルストア と DLR ランタイム コンポーネントを依存関係として使用することで、DLR とサンプルモデルをダウンロードします。

カスタムトレーニングされた DLR モデルでこの推論コンポーネントを使用するには、依存モデルストアコンポーネントのカスタムバージョンを作成します。独自のカスタム推論コードを使用するには、このコンポーネントの recipe をテンプレートとして使用して、[create a custom inference component] (カスタム推論コンポーネントを作成) できます。

トピック

- バージョン
- タイプ
- オペレーティングシステム
- 要件
- 依存関係
- 構成
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x
- 2.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Amazon Linux 2 または Ubuntu 18.04 を実行している Greengrass コアデバイスの場合は、GNU C ライブラリ (glibc) バージョン 2.27 以降がデバイスにインストールされている必要があります。
- Raspberry Pi などの Armv7l デバイスでは、OpenCV-Python の依存関係がデバイスにインストールされています。次のコマンドを実行して依存関係をインストールします。

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

- Raspberry Pi OS Bullseye を実行する Raspberry Pi デバイスでは、次の要件を満たす必要があります。
  - NumPy 1.22.4 以降がデバイスにインストールされている。Raspberry Pi OSwireseye には以前のバージョンの が含まれているため NumPy、次のコマンドを実行して NumPy デバイスでアップグレードできます。

    ```
    pip3 install --upgrade numpy
    ```

  - デバイスで、レガシーカメラスタックが有効になっていること。Raspberry Pi OS Bullseye には、デフォルトで新しいカメラスタックが含まれており有効化されていますが、これには互換性がないるため、レガシーカメラスタックを有効にしておく必要があります。

    レガシーカメラスタックを有効にするには

    1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

       ```
       sudo raspi-config
       ```

    2. [Interface Options] (インターフェイスオプション) を選択します。
    3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
    4. Raspberry Pi を再起動します。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネント

のリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.13

次の表に、このコンポーネントのバージョン 2.1.13 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト
DLR オブジェクト検出モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.12

次の表に、このコンポーネントのバージョン 2.1.12 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト
DLR オブジェクト検出モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.11

次の表に、このコンポーネントのバージョン 2.1.11 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト

依存関係	互換性のあるバージョン	依存関係タイプ
DLR オブジェクト検出モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.10

次の表に、このコンポーネントのバージョン 2.1.10 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト
DLR オブジェクト検出モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.9

次の表に、このコンポーネントのバージョン 2.1.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト
DLR オブジェクト検出モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.8

次の表に、このコンポーネントのバージョン 2.1.8 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト
DLR オブジェクト検出モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト
DLR オブジェクト検出モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.6

次の表に、このコンポーネントのバージョン 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト
DLR オブジェクト検出モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.4 - 2.1.5

次の表に、このコンポーネントのバージョン 2.1.4 から 2.1.5 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト
DLR オブジェクト検出モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト
DLR オブジェクト検出モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト
DLR オブジェクト検出モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト
DLR オブジェクト検出モデルストア	~2.1.0	ハード
DLR	~1.6.0	ハード

2.0.x

次の表に、このコンポーネントのバージョン 2.0.x の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	~2.0.0	ソフト
DLR オブジェクト検出モデルストア	~2.0.0	ハード
DLR	~1.3.0	ソフト

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

2.1.x

`accessControl`

(オプション) コンポーネントがデフォルトの通知トピックにメッセージをパブリッシュできるようにする承認ポリシーを含むオブジェクト。

デフォルト:

```
{
   "aws.greengrass.ipc.mqttproxy": {
      "aws.greengrass.DLRObjectDetection:mqttproxy:1": {
```

```
            "policyDescription": "Allows access to publish via topic ml/dlr/object-
detection.",
            "operations": [
               "aws.greengrass#PublishToIoTCore"
            ],
            "resources": [
               "ml/dlr/object-detection"
            ]
         }
      }
}
```

`PublishResultsOnTopic`

(オプション) 推論結果をパブリッシュするトピック。この値を変更する場合は、カスタムトピック名と一致するように `resources` パラメータの `accessControl` の値も変更する必要があります。

デフォルト: `ml/dlr/object-detection`

`Accelerator`

使用するアクセラレーター。サポートされている値は、`cpu` および `gpu` です。

依存モデルコンポーネントのサンプルモデルは CPU アクセラレーションのみをサポートします。別のカスタムモデルで GPU アクセラレーションを使用するには、カスタムモデルコンポーネントを作成して、パブリックモデルコンポーネントを上書きします。

デフォルト: `cpu`

`ImageDirectory`

(オプション) 推論コンポーネントがイメージを読み取る場所と成る、デバイス上のフォルダへのパス。この値は、読み取り/書き込みアクセス権を持つデバイスの任意の場所に変更できます。

デフォルト: *`/greengrass/v2`*`/packages/artifacts-unarchived/`*`component-name`*`/object_detection/sample_images/`

> **Note**
>
> `UseCamera` を `true` の値に設定した場合、この設定パラメータは無視されます。

`ImageName`

(オプション) 推論コンポーネントが予測を行う際の入力として使用するイメージの名前。コンポーネントは、`ImageDirectory` で指定されたフォルダ内のイメージを検索します。デフォルトでは、コンポーネントはデフォルトのイメージディレクトリにあるサンプルイメージを使用します。AWS IoT Greengrass は `jpeg`、`jpg`、`png`、および `npy` のイメージ形式をサポートします。

デフォルト: `objects.jpg`

> **Note**
>
> `UseCamera` を `true` の値に設定した場合、この設定パラメータは無視されます。

`InferenceInterval`

(オプション) 推論コードによって行われた各予測間の時間 (秒単位)。サンプル推論コードは無期限に実行され、指定された時間間隔で予測を繰り返します。例えば、カメラで撮影したイメージをリアルタイム予測に使用する場合などには、この間隔を短い間隔に変更できます。

デフォルト: `3600`

`ModelResourceKey`

(オプション) 依存パブリックモデルコンポーネントで使用されるモデル。このパラメータを変更するのは、パブリックモデルコンポーネントをカスタムコンポーネントでオーバーライドする場合のみです。

デフォルト:

```
{
    "armv7l": "DLR-yolo3-armv7l-cpu-ObjectDetection",
    "aarch64": "DLR-yolo3-aarch64-gpu-ObjectDetection",
    "x86_64": "DLR-yolo3-x86_64-cpu-ObjectDetection",
    "windows": "DLR-resnet50-win-cpu-ObjectDetection"
}
```

`UseCamera`

(オプション) Greengrass コアデバイスに接続されたカメラの画像を使用するかどうかを定義する文字列値。サポートされている値は、`true` および `false` です。

この値を `true` に設定した場合、サンプル推論コードはデバイスのカメラにアクセスし、キャプチャしたイメージでローカルに推論を実行します。`ImageName` と `ImageDirectory` パラメータの値は無視されます。このコンポーネントを実行しているユーザーが、カメラがキャプチャしたイメージを保存する場所への、読み込み/書き込みアクセス権を持っていることを確認します。

デフォルト: `false`

> **Note**
>
> このコンポーネントの recipe を表示すると、`UseCamera` 設定パラメータはデフォルト設定には表示されません。ただし、このパラメータの値は、コンポーネントをデプロイするときに、[configuration merge update] (設定マージの更新) で変更することができます。
>
> `UseCamera` を `true` に設定する場合は、ランタイムコンポーネントによって作成された仮想環境から推論コンポーネントがカメラにアクセスできるようにするためのシンボルリンクも作成する必要があります。サンプル推論コンポーネントを使用したカメラの使用方法の詳細については、「コンポーネント設定の更新」を参照してください。

2.0.x

`MLRootPath`

(オプション) 推論コンポーネントがイメージを読み取り、推論結果を書き込む Linux コアデバイスのフォルダのパス。この値は、このコンポーネントを実行しているユーザーが読み取り/書き込みアクセス権を持つデバイスの任意の場所に変更できます。

デフォルト: *`/greengrass/v2`*`/work/variant.DLR/greengrass_ml`

デフォルト: *`/greengrass/v2`*`/work/variant.TensorFlowLite/greengrass_ml`

`Accelerator`

変更しないでください。現在、アクセラレータでサポートされる値は `cpu` のみとなります。これは、依存モデルコンポーネントのモデルは CPU アクセラレータに対してのみコンパイルされるからです。

`ImageName`

(オプション) 推論コンポーネントが予測を行う際の入力として使用するイメージの名前。コンポーネントは、`ImageDirectory` で指定されたフォルダ内のイメージを検索します。デフォルトの場所は *`MLRootPath`*`/images` です。AWS IoT Greengrass は `jpeg`、`jpg`、`png`、`npy` イメージ形式をサポートしています。

デフォルト: `objects.jpg`

`InferenceInterval`

(オプション) 推論コードによって行われた各予測間の時間 (秒単位)。サンプル推論コードは無期限に実行され、指定された時間間隔で予測を繰り返します。例えば、カメラで撮影したイメージをリアルタイム予測に使用する場合などには、この間隔を短い間隔に変更できます。

デフォルト: `3600`

`ModelResourceKey`

(オプション) 依存パブリックモデルコンポーネントで使用されるモデル。このパラメータを変更するのは、パブリックモデルコンポーネントをカスタムコンポーネントでオーバーライドする場合のみです。

デフォルト:

```
{
    armv7l: "DLR-yolo3-armv7l-cpu-ObjectDetection",
    x86_64: "DLR-yolo3-x86_64-cpu-ObjectDetection"
}
```

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/aws.greengrass.DLRObjectDetection.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.DLRObjectDetection.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.DLRObjectDetection.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.greengrass.DLRObjectDetection.log -Tail 10
   -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.13	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.1.12	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.11	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.10	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.9	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.8	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.1.7	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.6	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.1.5	コンポーネントはすべての AWS リージョン にリリースされています。
2.1.4	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。 このバージョンは欧州 (ロンドン) では利用できません (`eu-west-2`)。
2.1.3	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.1.2	バグ修正と機能向上 • サンプルの DLR オブジェクト検出推論結果で不正確なバウンディングボックスが発生するイメージスケーリング上の問題を修正しました。

バージョン	変更
2.1.1	新機能 • 深層学習ランタイム のバージョン 1.6.0.を使用します。 • Armv8 (AArch64) プラットフォーム上でのサンプルオブジェクト検出のサポートが追加されました。これにより、Jetson Nano などの NVIDIA Jetson を実行している Greengrass コアデバイスにまで機械学習サポートが拡張されます。 • サンプル推論でカメラ統合が可能になりました。新しい `UseCamera` 設定パラメータを使用すると、サンプル推論コードが Greengrass コアデバイスのカメラにアクセスできるようになり、キャプチャしたイメージ上でローカルに推論を実行できるようになります。 • AWS クラウド に推論結果をパブリッシュするためのサポートが追加されました。新しい `PublishResultsOnTopic` 設定パラメータを使用して、結果をパブリッシュするトピックを指定します。 • 推論を実行するイメージ用のカスタムディレクトリを指定できる新しい `ImageDirectory` 設定パラメータが追加されました。 バグ修正と機能向上 • 別の推論ファイルではなく、コンポーネントログファイルに推論結果を書き込みます。 • AWS IoT Greengrass Core ソフトウェアのログモジュールを使用してコンポーネントの出力を記録します。 • AWS IoT Device SDK を使用してコンポーネント設定を読み込み、設定への変更を適用します。
2.0.4	当初のバージョン

DLR イメージ分類モデルストア

DLR イメージ分類モデルストアは、Greengrass アーティファクトとして事前トレーニング済みの ResNet-50 モデルを含む機械学習モデルコンポーネントです。このコンポーネントで使用される事前トレーニング済みモデルは GluonCV Model Zoo から取得され、 SageMaker Neo Deep Learning Runtime を使用してコンパイルされます。

DLR イメージ分類推論コンポーネントは、このコンポーネントをモデルソースの依存関係として使用します。カスタムトレーニングされた DLR モデルを使用するには、このモデルコンポーネントのカスタムバージョンを作成し、カスタムモデルをコンポーネントアーティファクトとして含めます。このコンポーネントの recipe をテンプレートとして使用して、カスタムモデルコンポーネントを作成できます。

> **Note**
>
> DLR イメージ分類モデルストアコンポーネントの名前は、そのバージョンによって異なります。2.1.x 以降のバージョンのコンポーネント名は `variant.DLR.ImageClassification.ModelStore` です。バージョン 2.0.x のコンポーネント名は `variant.ImageClassification.ModelStore` です。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x (`variant.DLR.ImageClassification.ModelStore`)
- 2.0.x (`variant.ImageClassification.ModelStore`)

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Amazon Linux 2 または Ubuntu 18.04 を実行している Greengrass コアデバイスの場合は、GNU C ライブラリ (glibc) バージョン 2.27 以降がデバイスにインストールされている必要があります。
- Raspberry Pi などの Armv7l デバイスでは、OpenCV-Python の依存関係がデバイスにインストールされています。次のコマンドを実行して依存関係をインストールします。

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

- Raspberry Pi OS Bullseye を実行する Raspberry Pi デバイスでは、次の要件を満たす必要があります。
  - NumPy 1.22.4 以降がデバイスにインストールされている。Raspberry Pi OSwireseye には以前のバージョンの が含まれているため NumPy、次のコマンドを実行して NumPy デバイスでアップグレードできます。

    ```
    pip3 install --upgrade numpy
    ```

  - デバイスで、レガシーカメラスタックが有効になっていること。Raspberry Pi OS Bullseye には、デフォルトで新しいカメラスタックが含まれており有効化されていますが、これには互換性がないるため、レガシーカメラスタックを有効にしておく必要があります。

    レガシーカメラスタックを有効にするには

    1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

       ```
       sudo raspi-config
       ```

2. [Interface Options] (インターフェイスオプション) を選択します。
3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
4. Raspberry Pi を再起動します。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.12

次の表に、このコンポーネントのバージョン 2.1.12 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト

2.1.11

次の表に、このコンポーネントのバージョン 2.1.11 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト

2.1.10

次の表に、このコンポーネントのバージョン 2.1.10 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト

2.1.9

次の表に、このコンポーネントのバージョン 2.1.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト

2.1.8

次の表に、このコンポーネントのバージョン 2.1.8 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト

2.1.6

次の表に、このコンポーネントのバージョン 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト

2.1.5

次の表に、このコンポーネントのバージョン 2.1.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト

2.1.4

次の表に、このコンポーネントのバージョン 2.1.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト

2.0.x

次の表に、このコンポーネントのバージョン 2.0.x の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	~2.0.0	ソフト

構成

このコンポーネントに設定パラメータはありません。

ローカルログファイル

このコンポーネントはログを出力しません。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.12	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.1.11	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.10	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.9	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.8	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.7	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.1.6	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.5	新機能 • Windows コアデバイスのサンプルイメージ分類モデルを追加しました。 • Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.1.4	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.1.3	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.1.1	新機能 • Armv8 (AArch64) プラットフォーム用のサンプル ResNet-50 イメージ分類モデルを追加します。これにより、Jetson Nano などの NVIDIA Jetson を実行している Greengrass コアデバイスにまで機械学習サポートが拡張されます。
2.0.4	当初のバージョン

DLR オブジェクト検出モデルストア

DLR オブジェクト検出モデルストアは、Greengrass アーティファクトとして事前学習済みの YOLOv3 モデルを含む機械学習モデルコンポーネントです。このコンポーネントで使用されるサンプルモデルは、 GluonCV Model Zoo から取得され、 SageMaker Neo Deep Learning Runtime を使用してコンパイルされます。

DLR オブジェクト検出推論コンポーネントは、このコンポーネントをモデルソースの依存関係として使用します。カスタムトレーニングされた DLR モデルを使用するには、このモデルコンポーネントの カスタムバージョンを作成し、カスタムモデルをコンポーネントアーティファクトとして含め

ます。このコンポーネントの recipe をテンプレートとして使用して、カスタムモデルコンポーネントを作成できます。

> **Note**
>
> DLR オブジェクト検出モデルストアコンポーネントの名前は、そのバージョンによって異なります。2.1.x 以降のバージョンのコンポーネント名は `variant.DLR.ObjectDetection.ModelStore` です。バージョン 2.0.x のコンポーネント名は `variant.ObjectDetection.ModelStore` です。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x
- 2.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Amazon Linux 2 または Ubuntu 18.04 を実行している Greengrass コアデバイスの場合は、GNU C ライブラリ (glibc) バージョン 2.27 以降がデバイスにインストールされている必要があります。
- Raspberry Pi などの Armv7l デバイスでは、OpenCV-Python の依存関係がデバイスにインストールされています。次のコマンドを実行して依存関係をインストールします。

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

- Raspberry Pi OS Bullseye を実行する Raspberry Pi デバイスでは、次の要件を満たす必要があります。
  - NumPy 1.22.4 以降がデバイスにインストールされている。Raspberry Pi OS Bullseye には以前のバージョンの が含まれているため NumPy、次のコマンドを実行して NumPy デバイスでアップグレードできます。

    ```
    pip3 install --upgrade numpy
    ```

  - デバイスで、レガシーカメラスタックが有効になっていること。Raspberry Pi OS Bullseye には、デフォルトで新しいカメラスタックが含まれており有効化されていますが、これには互換性がないるため、レガシーカメラスタックを有効にしておく必要があります。

    レガシーカメラスタックを有効にするには

    1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

       ```
       sudo raspi-config
       ```

    2. [Interface Options] (インターフェイスオプション) を選択します。

3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
4. Raspberry Pi を再起動します。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.13

次の表に、このコンポーネントのバージョン 2.1.13 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト

2.1.12

次の表に、このコンポーネントのバージョン 2.1.12 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト

2.1.11

次の表に、このコンポーネントのバージョン 2.1.11 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト

2.1.10

次の表に、このコンポーネントのバージョン 2.1.10 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト

2.1.9

次の表に、このコンポーネントのバージョン 2.1.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト

2.1.8

次の表に、このコンポーネントのバージョン 2.1.8 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト

2.1.5 and 2.1.6

次の表に、このコンポーネントのバージョン 2.1.5 および 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト

2.1.4

次の表に、このコンポーネントのバージョン 2.1.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト

2.0.x

次の表に、このコンポーネントのバージョン 2.0.x の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	~2.0.0	ソフト

構成

このコンポーネントに設定パラメータはありません。

ローカルログファイル

このコンポーネントはログを出力しません。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.13	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.1.12	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.11	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.10	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.9	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.8	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.1.7	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.6	Armv8 (Aarch64) デバイス上の問題を修正する CPU モデルを追加しました。
2.1.5	新機能 • Windows コアデバイス用のサンプルオブジェクト検出モデルを追加しました。 バグ修正と機能向上 • Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.1.4	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.1.3	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.1.1	新機能 • Armv8 (aArch64) プラットフォーム用のサンプル YoLov3 オブジェクト検出モデルが追加されました。これにより、Jetson Nano などの NVIDIA Jetson を実行している Greengrass コアデバイスにまで機械学習サポートが拡張されます。
2.0.4	当初のバージョン

DLR ランタイム

DLR ランタイムコンポーネント (`variant.DLR`) には、深層学習ランタイム (DLR) とその依存関係をお使いのデバイスの仮想環境にインストールするスクリプトが含まれています。DLR イメージ分類 および DLR オブジェクトの検出 コンポーネントは、DLR をインストールするための依存関係

としてこのコンポーネントを使用します。コンポーネントバージョン 1.6.x は DLR v1.6.0 をインストールし、コンポーネントバージョン 1.3.x は DLR v1.3.0 をインストールします。

別のランタイムを使用するには、このコンポーネントの recipe をテンプレートとして使用して、カスタム機械学習コンポーネントを作成することができます。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 1.6.x
- 1.3.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux

- Windows

要件

このコンポーネントには次の要件があります。

- Amazon Linux 2 または Ubuntu 18.04 を実行している Greengrass コアデバイスの場合は、GNU C ライブラリ (glibc) バージョン 2.27 以降がデバイスにインストールされている必要があります。
- Raspberry Pi などの Armv7l デバイスでは、OpenCV-Python の依存関係がデバイスにインストールされています。次のコマンドを実行して依存関係をインストールします。

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

- Raspberry Pi OS Bullseye を実行する Raspberry Pi デバイスでは、次の要件を満たす必要があります。
  - NumPy 1.22.4 以降がデバイスにインストールされている。Raspberry Pi OSwireseye には以前のバージョンの が含まれているため NumPy、次のコマンドを実行して NumPy デバイスでアップグレードできます。

    ```
    pip3 install --upgrade numpy
    ```

  - デバイスで、レガシーカメラスタックが有効になっていること。Raspberry Pi OS Bullseye には、デフォルトで新しいカメラスタックが含まれており有効化されていますが、これには互換性がないるため、レガシーカメラスタックを有効にしておく必要があります。

    レガシーカメラスタックを有効にするには

    1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

       ```
       sudo raspi-config
       ```

    2. [Interface Options] (インターフェイスオプション) を選択します。
    3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
    4. Raspberry Pi を再起動します。

エンドポイントおよびポート

デフォルトでは、このコンポーネントは、インストーラスクリプトを使い、コアデバイスが使用するプラットフォームに応じて、apt、yum、brew、および pip コマンドを使用してパッケージをインストールします。このコンポーネントは、インストーラスクリプトを実行するために、さまざまなパッケージインデックスおよびリポジトリへのアウトバウンドリクエストを実行できる必要があります。このコンポーネントのアウトバウンドトラフィックがプロキシまたはファイアウォールを通過できるようにするには、コアデバイスがインストールに接続するパッケージインデックスとリポジトリのエンドポイントを特定する必要があります。

このコンポーネントのインストールスクリプトに必要なエンドポイントを特定するときは、次の点を考慮してください。

- エンドポイントは、コアデバイスのプラットフォームによって異なります。例えば、Ubuntu を実行するコアデバイスでは、yum または brew ではなく apt を使用します。さらに、同じパッケージインデックスを使用するデバイスは、異なるソースリストを持つ可能性があるため、異なるリポジトリからパッケージを取得する場合があります。
- 各デバイスにはパッケージの取得場所を定義する独自のソースリストがあるため、同じパッケージインデックスを使用する複数のデバイス間でエンドポイントが異なる場合があります。
- エンドポイントは時間の経過とともに変化する可能性があります。各パッケージインデックスは、パッケージをダウンロードするリポジトリの URL を提供し、パッケージの所有者は、パッケージインデックスが提供する URL を変更できます。

このコンポーネントがインストールする依存関係とインストーラスクリプトを無効にする方法の詳細については、「 UseInstaller設定パラメータ」を参照してください。

基本的な操作に必要なエンドポイントとポートの詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

1.6.11 and 1.6.12

次の表に、このコンポーネントのバージョン 1.6.11 および 1.6.12 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <3.0.0	ソフト

1.6.10

次の表に、このコンポーネントのバージョン 1.6.10 における依存関係の一覧を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト

1.6.9

次の表に、このコンポーネントのバージョン 1.6.9 における依存関係の一覧を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト

1.6.8

次の表に、このコンポーネントのバージョン 1.6.8 における依存関係の一覧を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト

1.6.6 and 1.6.7

次の表に、このコンポーネントのバージョン 1.6.6 および 1.6.7 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト

1.6.4 and 1.6.5

次の表に、このコンポーネントのバージョン 1.6.4 および 1.6.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト

1.6.3

次の表に、このコンポーネントのバージョン 1.6.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト

1.6.2

次の表に、このコンポーネントのバージョン 1.6.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト

1.6.1

次の表に、このコンポーネントのバージョン 1.6.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト

1.3.x

次の表に、このコンポーネントのバージョン 1.3.x の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	~2.0.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

`MLRootPath`

(オプション) 推論コンポーネントがイメージを読み取り、推論結果を書き込む Linux コアデバイスのフォルダのパス。この値は、このコンポーネントを実行しているユーザーが読み取り/書き込みアクセス権を持つデバイスの任意の場所に変更できます。

デフォルト: */greengrass/v2*/work/variant.DLR/greengrass_ml

`WindowsMLRootPath`

この機能は、このコンポーネントの v1.6.6 以降で利用できます。

(オプション) 推論コンポーネントがイメージを読み取り、推論結果を書き込む Windows コアデバイスのフォルダのパス。この値は、このコンポーネントを実行しているユーザーが読み取り/書き込みアクセス権を持つデバイスの任意の場所に変更できます。

デフォルト: *C:\greengrass\v2*\\work\\variant.DLR\\greengrass_ml

`UseInstaller`

(オプション) DLR とその依存関係をインストールするために、このコンポーネントでインストーラスクリプトを使用するかどうかを定義する文字列値。サポートされている値は、`true` および `false` です。

DLR のインストールにカスタムスクリプトを使用する場合、またはビルド済みの Linux イメージにランタイムの依存関係を含める場合は、この値を `false` に設定します。このコンポーネント

を AWS の提供する DLR 推論コンポーネントで使用するには、依存関係を含む次のライブラリをインストールし、ML コンポーネントを実行する `ggc_user` などのシステムユーザーを利用できるようにします。

- Python 3.7 以降 (ご使用のバージョンの Python 用の `pip` を含む)。
- 深層学習ランタイムのバージョン 1.6.0
- NumPy.
- OpenCV-Python。
- AWS IoT Device SDK v2 for Python。
- AWS 共通ランタイム (CRT) Python。
- Picamera (Raspberry Pi デバイスのみ)。
- `awscam` モジュール (AWS DeepLens デバイス用)。
- libGL (Linux デバイス用)

デフォルト: `true`

使用方法

このコンポーネントを `UseInstaller` 設定パラメータを `true` 設定した状態で使用し、DLR とその依存関係をデバイスにインストールします。コンポーネントは、DLR に必要な OpenCV と NumPy ライブラリを含む仮想環境をデバイスに設定します。

> **Note**
>
> このコンポーネントのインストーラスクリプトは、デバイスの仮想環境を設定し、インストールされている機械学習フレームワークを使用するために必要な最新バージョンの追加システムライブラリもインストールします。これにより、デバイスで既存のシステムライブラリがアップグレードされる可能性があります。次の表で、サポートされている各オペレーティングシステムに、このコンポーネントがインストールするライブラリの一覧を確認してください。このインストールプロセスをカスタマイズする場合は、`UseInstaller` 設定パラメータを `false` に設定し、独自のインストーラスクリプトを開発します。

プラットフォーム	デバイスシステムにインストールされているライブラリ	仮想環境にインストールされているライブラリ
Armv7l	`build-essential` , `cmake`, `ca-certificates` , `git`	`setuptools` , `wheel`
Amazon Linux 2	`mesa-libGL`	なし
Ubuntu	`wget`	なし

推論コンポーネントをデプロイすると、このランタイムコンポーネントはまず、デバイスにすでに DLR とその依存関係がインストールされているかどうかを検証し、インストールされていない場合にはインストールします。

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/variant.DLR.log
```

Windows

```
C:\greengrass\v2\logs\variant.DLR.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/variant.DLR.log
  ```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\variant.DLR.log -Tail 10 -Wait
```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
1.6.12	バグ修正と機能向上 • Windows OS ユーザー用のインストールスクリプトを修正します。
1.6.11	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
1.6.10	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
1.6.9	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
1.6.8	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
1.6.7	バグ修正と機能向上 • `UseInstaller` インストールスクリプトを更新して libGL をインストールします。これは、一部の Linux プラットフォームではデフォルトで利用できません。 • このコンポーネントの仮想環境で常に Python 3.9 を使用するように、`UseInstaller` インストールスクリプトを更新します。この変更は、他のライブラリとの互換性を確保するのに役立ちます。
1.6.6	新機能 • Windows を実行するコアデバイスのサポートが追加されました。

バージョン	変更
	• Windows コアデバイスで推論結果フォルダを設定するために使用できる新しい `WindowsMLRootPath` 設定パラメータを追加しました。
1.6.5	新機能 • このコンポーネントのインストールスクリプトを無効にできる新しい `UseInstaller` 設定パラメータを追加しました。
1.6.4	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
1.6.3	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
1.6.2	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
1.6.1	新機能 • 深層学習ランタイム v1.6.0 とその依存関係をインストールします。 • Armv8 (AArch64) プラットフォームへの DLR のインストールに対するサポートが追加されました。これにより、Jetson Nano などの NVIDIA Jetson を実行している Greengrass コアデバイスにまで機械学習サポートが拡張されます。 バグ修正と機能向上 • 仮想環境で AWS IoT Device SDK をインストールしてコンポーネント設定を読み取り、設定の変更を適用します。 • マイナーなバグの追加修正と機能向上。
1.3.2	当初のバージョン DLR v1.3.0 をインストールします。

TensorFlow Lite イメージ分類

TensorFlow Lite イメージ分類コンポーネント (`aws.greengrass.TensorFlowLiteImageClassification`) には、TensorFlow Lite ランタイムと事前トレーニング済みの MobileNet 1.0 量子化モデルのサンプルを使用してイメージ分類推論を実行するためのサンプル推論コードが含まれています。このコンポーネントは、バリアン

トTensorFlow Lite イメージ分類モデルストアとTensorFlow Lite ランタイムコンポーネントを依存関係として使用し、 TensorFlow Lite ランタイムとサンプルモデルをダウンロードします。

この推論コンポーネントをカスタムトレーニング済み TensorFlow Lite モデルで使用するには、依存モデルストアコンポーネントのカスタムバージョンを作成します。独自のカスタム推論コードを使用するには、このコンポーネントの recipe をテンプレートとして使用して、[create a custom inference component] (カスタム推論コンポーネントを作成) できます。

トピック

- バージョン
- タイプ
- オペレーティングシステム
- 要件
- 依存関係
- 構成
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Amazon Linux 2 または Ubuntu 18.04 を実行している Greengrass コアデバイスの場合は、GNU C ライブラリ (glibc) バージョン 2.27 以降がデバイスにインストールされている必要があります。
- Raspberry Pi などの Armv7l デバイスでは、OpenCV-Python の依存関係がデバイスにインストールされています。次のコマンドを実行して依存関係をインストールします。

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

- Raspberry Pi OS Bullseye を実行する Raspberry Pi デバイスでは、次の要件を満たす必要があります。
  - NumPy 1.22.4 以降がデバイスにインストールされている。Raspberry Pi OSwireseye には以前のバージョンの が含まれているため NumPy、次のコマンドを実行して NumPy デバイスでアップグレードできます。

    ```
    pip3 install --upgrade numpy
    ```

  - デバイスで、レガシーカメラスタックが有効になっていること。Raspberry Pi OS Bullseye には、デフォルトで新しいカメラスタックが含まれており有効化されていますが、これには互換性がないるため、レガシーカメラスタックを有効にしておく必要があります。

    レガシーカメラスタックを有効にするには

    1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

       ```
       sudo raspi-config
       ```

    2. [Interface Options] (インターフェイスオプション) を選択します。
    3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
    4. Raspberry Pi を再起動します。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネント

のリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.11

次の表に、このコンポーネントのバージョン 2.1.11 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.10

次の表に、このコンポーネントのバージョン 2.1.10 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.9

次の表に、このコンポーネントのバージョン 2.1.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト

依存関係	互換性のあるバージョン	依存関係タイプ
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.8

次の表に、このコンポーネントのバージョン 2.1.8 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.6

次の表に、このコンポーネントのバージョン 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.5

次の表に、このコンポーネントのバージョン 2.1.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.4

次の表に、このコンポーネントのバージョン 2.1.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.0

次の表に、このコンポーネントのバージョン 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

accessControl

(オプション) コンポーネントがデフォルトの通知トピックにメッセージをパブリッシュできるようにする承認ポリシーを含むオブジェクト。

デフォルト:

```
{
   "aws.greengrass.ipc.mqttproxy": {
      "aws.greengrass.TensorFlowLiteImageClassification:mqttproxy:1": {
         "policyDescription": "Allows access to publish via topic ml/tflite/image-
classification.",
         "operations": [
            "aws.greengrass#PublishToIoTCore"
         ],
         "resources": [
            "ml/tflite/image-classification"
         ]
      }
   }
}
```

PublishResultsOnTopic

(オプション) 推論結果をパブリッシュするトピック。この値を変更する場合は、カスタムトピック名と一致するように resources パラメータの accessControl の値も変更する必要があります。

デフォルト: `ml/tflite/image-classification`

`Accelerator`

使用するアクセラレーター。サポートされている値は、`cpu` および `gpu` です。

依存モデルコンポーネントのサンプルモデルは CPU アクセラレーションのみをサポートします。別のカスタムモデルで GPU アクセラレーションを使用するには、[カスタムモデルコンポーネントを作成]して、パブリックモデルコンポーネントを上書きします。

デフォルト: `cpu`

`ImageDirectory`

(オプション) 推論コンポーネントがイメージを読み取る場所と成る、デバイス上のフォルダへのパス。この値は、読み取り/書き込みアクセス権を持つデバイスの任意の場所に変更できます。

デフォルト: *`/greengrass/v2`*`/packages/artifacts-unarchived/`*`component-name`*`/image_classification/sample_images/`

> **Note**
>
> `UseCamera` を `true` の値に設定した場合、この設定パラメータは無視されます。

`ImageName`

(オプション) 推論コンポーネントが予測を行う際の入力として使用するイメージの名前。コンポーネントは、`ImageDirectory` で指定されたフォルダ内のイメージを検索します。デフォルトでは、コンポーネントはデフォルトのイメージディレクトリにあるサンプルイメージを使用します。AWS IoT Greengrass は `jpeg`、`jpg`、`png`、および `npy` のイメージ形式をサポートします。

デフォルト: `cat.jpeg`

> **Note**
>
> `UseCamera` を `true` の値に設定した場合、この設定パラメータは無視されます。

`InferenceInterval`

(オプション) 推論コードによって行われた各予測間の時間 (秒単位)。サンプル推論コードは無期限に実行され、指定された時間間隔で予測を繰り返します。例えば、カメラで撮影したイメージをリアルタイム予測に使用する場合などには、この間隔を短い間隔に変更できます。

デフォルト: `3600`

`ModelResourceKey`

(オプション) 依存パブリックモデルコンポーネントで使用されるモデル。このパラメータを変更するのは、パブリックモデルコンポーネントをカスタムコンポーネントでオーバーライドする場合のみです。

デフォルト:

```
{
    "model": "TensorFlowLite-Mobilenet"
}
```

`UseCamera`

(オプション) Greengrass コアデバイスに接続されたカメラの画像を使用するかどうかを定義する文字列値。サポートされている値は、`true` および `false` です。

この値を `true` に設定した場合、サンプル推論コードはデバイスのカメラにアクセスし、キャプチャしたイメージでローカルに推論を実行します。`ImageName` と `ImageDirectory` パラメータの値は無視されます。このコンポーネントを実行しているユーザーが、カメラがキャプチャしたイメージを保存する場所への、読み込み/書き込みアクセス権を持っていることを確認します。

デフォルト: `false`

> **Note**
>
> このコンポーネントの recipe を表示すると、`UseCamera` 設定パラメータはデフォルト設定には表示されません。ただし、このパラメータの値は、コンポーネントをデプロイするときに、[configuration merge update] (設定マージの更新) で変更することができます。`UseCamera` を `true` に設定する場合は、ランタイムコンポーネントによって作成された仮想環境から推論コンポーネントがカメラにアクセスできるようにするためのシンボルリンクも作成する必要があります。サンプル推論コンポーネントを使用したカメラの使用方法の詳細については、「コンポーネント設定の更新」を参照してください。

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/aws.greengrass.TensorFlowLiteImageClassification.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.TensorFlowLiteImageClassification.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/
  aws.greengrass.TensorFlowLiteImageClassification.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs
  \aws.greengrass.TensorFlowLiteImageClassification.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.11	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。

バージョン	変更
2.1.10	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.9	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.8	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.7	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.6	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.1.5	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.4	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.1.3	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.1.1	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.1.0	当初のバージョン

TensorFlow ライトオブジェクト検出

TensorFlow Lite オブジェクト検出コンポーネント (`aws.greengrass.TensorFlowLiteObjectDetection`) には、TensorFlow Lite を使用してオブジェクト検出推論を実行するためのサンプル推論コードと、事前トレーニング済みのサンプルシン

グルショット検出 (SSD) MobileNet 1.0 モデルが含まれています。このコンポーネントは、バリアントTensorFlow Lite オブジェクト検出モデルストアとTensorFlow Lite ランタイムコンポーネントを依存関係として使用し、 TensorFlow Lite とサンプルモデルをダウンロードします。

この推論コンポーネントをカスタムトレーニング済み TensorFlow Lite モデルで使用するには、依存モデルストアコンポーネントのカスタムバージョンを作成できます。独自のカスタム推論コードを使用するには、このコンポーネントの recipe をテンプレートとして使用して、[create a custom inference component] (カスタム推論コンポーネントを作成) します。

トピック

- バージョン
- タイプ
- オペレーティングシステム
- 要件
- 依存関係
- 構成
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Amazon Linux 2 または Ubuntu 18.04 を実行している Greengrass コアデバイスの場合は、GNU C ライブラリ (glibc) バージョン 2.27 以降がデバイスにインストールされている必要があります。
- Raspberry Pi などの Armv7l デバイスでは、OpenCV-Python の依存関係がデバイスにインストールされています。次のコマンドを実行して依存関係をインストールします。

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

- Raspberry Pi OS Bullseye を実行する Raspberry Pi デバイスでは、次の要件を満たす必要があります。
  - NumPy 1.22.4 以降がデバイスにインストールされている。Raspberry Pi OS Bullseye には以前のバージョンの が含まれているため NumPy、次のコマンドを実行して NumPy デバイスでアップグレードできます。

    ```
    pip3 install --upgrade numpy
    ```

  - デバイスで、レガシーカメラスタックが有効になっていること。Raspberry Pi OS Bullseye には、デフォルトで新しいカメラスタックが含まれており有効化されていますが、これには互換性がないるため、レガシーカメラスタックを有効にしておく必要があります。

    レガシーカメラスタックを有効にするには

    1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

       ```
       sudo raspi-config
       ```

    2. [Interface Options] (インターフェイスオプション) を選択します。
    3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
    4. Raspberry Pi を再起動します。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.11

次の表に、このコンポーネントのバージョン 2.1.11 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.10

次の表に、このコンポーネントのバージョン 2.1.10 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.9

次の表に、このコンポーネントのバージョン 2.1.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.8

次の表に、このコンポーネントのバージョン 2.1.8 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.6

次の表に、このコンポーネントのバージョン 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.5

次の表に、このコンポーネントのバージョン 2.1.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.4

次の表に、このコンポーネントのバージョン 2.1.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

2.1.0

次の表に、このコンポーネントのバージョン 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト
TensorFlow Lite イメージ分類モデルストア	>=2.1.0 <2.2.0	ハード
TensorFlow ライト	>=2.5.0 <2.6.0	ハード

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

accessControl

(オプション) コンポーネントがデフォルトの通知トピックにメッセージをパブリッシュできるようにする承認ポリシーを含むオブジェクト。

デフォルト:

```
{
   "aws.greengrass.ipc.mqttproxy": {
      "aws.greengrass.TensorFlowLiteObjectDetection:mqttproxy:1": {
         "policyDescription": "Allows access to publish via topic ml/tflite/object-
detection.",
         "operations": [
            "aws.greengrass#PublishToIoTCore"
         ],
         "resources": [
            "ml/tflite/object-detection"
         ]
      }
   }
}
```

PublishResultsOnTopic

(オプション) 推論結果をパブリッシュするトピック。この値を変更する場合は、カスタムトピック名と一致するように resources パラメータの accessControl の値も変更する必要があります。

デフォルト: `ml/tflite/object-detection`

`Accelerator`

使用するアクセラレーター。サポートされている値は、`cpu` および `gpu` です。

依存モデルコンポーネントのサンプルモデルは CPU アクセラレーションのみをサポートします。別のカスタムモデルで GPU アクセラレーションを使用するには、[カスタムモデルコンポーネントを作成]して、パブリックモデルコンポーネントを上書きします。

デフォルト: `cpu`

`ImageDirectory`

(オプション) 推論コンポーネントがイメージを読み取る場所と成る、デバイス上のフォルダへのパス。この値は、読み取り/書き込みアクセス権を持つデバイスの任意の場所に変更できます。

デフォルト: *`/greengrass/v2`*`/packages/artifacts-unarchived/`*`component-name`*`/object_detection/sample_images/`

> **Note**
>
> `UseCamera` を `true` の値に設定した場合、この設定パラメータは無視されます。

`ImageName`

(オプション) 推論コンポーネントが予測を行う際の入力として使用するイメージの名前。コンポーネントは、`ImageDirectory` で指定されたフォルダ内のイメージを検索します。デフォルトでは、コンポーネントはデフォルトのイメージディレクトリにあるサンプルイメージを使用します。AWS IoT Greengrass は `jpeg`、`jpg`、`png`、および `npy` のイメージ形式をサポートします。

デフォルト: `objects.jpg`

> **Note**
>
> `UseCamera` を `true` の値に設定した場合、この設定パラメータは無視されます。

`InferenceInterval`

(オプション) 推論コードによって行われた各予測間の時間 (秒単位)。サンプル推論コードは無期限に実行され、指定された時間間隔で予測を繰り返します。例えば、カメラで撮影したイメージをリアルタイム予測に使用する場合などには、この間隔を短い間隔に変更できます。

デフォルト: `3600`

`ModelResourceKey`

(オプション) 依存パブリックモデルコンポーネントで使用されるモデル。このパラメータを変更するのは、パブリックモデルコンポーネントをカスタムコンポーネントでオーバーライドする場合のみです。

デフォルト:

```
{
    "model": "TensorFlowLite-SSD"
}
```

`UseCamera`

(オプション) Greengrass コアデバイスに接続されたカメラの画像を使用するかどうかを定義する文字列値。サポートされている値は、`true` および `false` です。

この値を `true` に設定した場合、サンプル推論コードはデバイスのカメラにアクセスし、キャプチャしたイメージでローカルに推論を実行します。`ImageName` と `ImageDirectory` パラメータの値は無視されます。このコンポーネントを実行しているユーザーが、カメラがキャプチャしたイメージを保存する場所への、読み込み/書き込みアクセス権を持っていることを確認します。

デフォルト: `false`

> **Note**
>
> このコンポーネントの recipe を表示すると、`UseCamera` 設定パラメータはデフォルト設定には表示されません。ただし、このパラメータの値は、コンポーネントをデプロイするときに、[configuration merge update] (設定マージの更新) で変更することができます。`UseCamera` を `true` に設定する場合は、ランタイムコンポーネントによって作成された仮想環境から推論コンポーネントがカメラにアクセスできるようにするためのシンボルリンクも作成する必要があります。サンプル推論コンポーネントを使用したカメラの使用方法の詳細については、「コンポーネント設定の更新」を参照してください。

> **Note**
>
> このコンポーネントの recipe を表示すると、`UseCamera` 設定パラメータはデフォルト設定には表示されません。ただし、このパラメータの値は、コンポーネントをデプロイするときに、[configuration merge update] (設定マージの更新) で変更することができます。`UseCamera` を `true` に設定する場合は、ランタイムコンポーネントによって作成された仮想環境から推論コンポーネントがカメラにアクセスできるようにするためのシンボルリンクも作成する必要があります。サンプル推論コンポーネントを使用したカメラの使用方法の詳細については、「コンポーネント設定の更新」を参照してください。

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/aws.greengrass.TensorFlowLiteObjectDetection.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.TensorFlowLiteObjectDetection.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/
  aws.greengrass.TensorFlowLiteObjectDetection.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs
  \aws.greengrass.TensorFlowLiteObjectDetection.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.11	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.1.10	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.9	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.8	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.7	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.6	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.1.5	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.4	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.1.3	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.1.1	バグ修正と機能向上 • サンプル TensorFlow Lite オブジェクト検出推論結果で不正確な境界ボックスが発生するイメージスケーリングの問題を修正しました。

バージョン	変更
2.1.0	当初のバージョン

TensorFlow Lite イメージ分類モデルストア

TensorFlow Lite イメージ分類モデルストア (`variant.TensorFlowLite.ImageClassification.ModelStore`) は、Greengrass アーティファクトとして事前トレーニング済みの MobileNet v1 モデルを含む機械学習モデルコンポーネントです。このコンポーネントで使用されるサンプルモデルは、TensorFlowHub から取得され、TensorFlow Lite を使用して実装されます。

TensorFlow Lite イメージ分類 推論コンポーネントは、このコンポーネントをモデルソースの依存関係として使用します。カスタムトレーニング済み TensorFlow Lite モデルを使用するには、このモデルコンポーネントのカスタムバージョンを作成し、カスタムモデルをコンポーネントアーティファクトとして含めます。このコンポーネントの recipe をテンプレートとして使用して、カスタムモデルコンポーネントを作成できます。

トピック

- バージョン
- タイプ
- オペレーティングシステム
- 要件
- 依存関係
- 構成
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Amazon Linux 2 または Ubuntu 18.04 を実行している Greengrass コアデバイスの場合は、GNU C ライブラリ (glibc) バージョン 2.27 以降がデバイスにインストールされている必要があります。
- Raspberry Pi などの Armv7l デバイスでは、OpenCV-Python の依存関係がデバイスにインストールされています。次のコマンドを実行して依存関係をインストールします。

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

- Raspberry Pi OS Bullseye を実行する Raspberry Pi デバイスでは、次の要件を満たす必要があります。
  - NumPy 1.22.4 以降がデバイスにインストールされている。Raspberry Pi OSwireseye には以前のバージョンの が含まれているため NumPy、次のコマンドを実行して NumPy デバイスでアップグレードできます。

    ```
    pip3 install --upgrade numpy
    ```

  - デバイスで、レガシーカメラスタックが有効になっていること。Raspberry Pi OS Bullseye には、デフォルトで新しいカメラスタックが含まれており有効化されていますが、これには互換性がないため、レガシーカメラスタックを有効にしておく必要があります。

レガシーカメラスタックを有効にするには

1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

```
sudo raspi-config
```

2. [Interface Options] (インターフェイスオプション) を選択します。
3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
4. Raspberry Pi を再起動します。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.11

次の表に、このコンポーネントのバージョン 2.1.11 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト

2.1.10

次の表に、このコンポーネントのバージョン 2.1.10 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト

2.1.9

次の表に、このコンポーネントのバージョン 2.1.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト

2.1.8

次の表に、このコンポーネントのバージョン 2.1.8 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト

2.1.6

次の表に、このコンポーネントのバージョン 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト

2.1.5

次の表に、このコンポーネントのバージョン 2.1.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト

2.1.4

次の表に、このコンポーネントのバージョン 2.1.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト

2.1.0

次の表に、このコンポーネントのバージョン 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト

構成

このコンポーネントに設定パラメータはありません。

ローカルログファイル

このコンポーネントはログを出力しません。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.11	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.1.10	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.9	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.8	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.7	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.6	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.1.5	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.4	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.1.3	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.1.1	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.1.0	当初のバージョン

TensorFlow Lite オブジェクト検出モデルストア

TensorFlow Lite オブジェクト検出モデルストア (`variant.TensorFlowLite.ObjectDetection.ModelStore`) は、Greengrass アーティファクトとして事前トレーニング済みのシングルショット検出 (SSD) モデルを含む機械学習 MobileNet モデルコンポーネントです。このコンポーネントで使用されるサンプルモデルは、TensorFlow Hub から取得され、TensorFlow Lite を使用して実装されます。

TensorFlow Lite オブジェクト検出推論コンポーネントは、このコンポーネントをモデルソースの依存関係として使用します。カスタムトレーニング済み TensorFlow Lite モデルを使用するには、このモデルコンポーネントのカスタムバージョンを作成し、カスタムモデルをコンポーネントアーティファクトとして含めます。このコンポーネントの recipe をテンプレートとして使用して、カスタムモデルコンポーネントを作成できます。

トピック

- バージョン
- タイプ
- オペレーティングシステム
- 要件

- 依存関係
- 構成
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Amazon Linux 2 または Ubuntu 18.04 を実行している Greengrass コアデバイスの場合は、GNU C ライブラリ (glibc) バージョン 2.27 以降がデバイスにインストールされている必要があります。
- Raspberry Pi などの Armv7l デバイスでは、OpenCV-Python の依存関係がデバイスにインストールされています。次のコマンドを実行して依存関係をインストールします。

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
 libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

- Raspberry Pi OS Bullseye を実行する Raspberry Pi デバイスでは、次の要件を満たす必要があります。
  - NumPy 1.22.4 以降がデバイスにインストールされている。Raspberry Pi OS Bullseye には以前のバージョンの が含まれているため NumPy、次のコマンドを実行して NumPy デバイスでアップグレードできます。

    ```
    pip3 install --upgrade numpy
    ```

  - デバイスで、レガシーカメラスタックが有効になっていること。Raspberry Pi OS Bullseye には、デフォルトで新しいカメラスタックが含まれており有効化されていますが、これには互換性がないるため、レガシーカメラスタックを有効にしておく必要があります。

    レガシーカメラスタックを有効にするには

    1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

       ```
       sudo raspi-config
       ```

    2. [Interface Options] (インターフェイスオプション) を選択します。
    3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
    4. Raspberry Pi を再起動します。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.11

次の表に、このコンポーネントのバージョン 2.1.11 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト

2.1.10

次の表に、このコンポーネントのバージョン 2.1.10 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト

2.1.9

次の表に、このコンポーネントのバージョン 2.1.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト

2.1.8

次の表に、このコンポーネントのバージョン 2.1.8 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト

2.1.6

次の表に、このコンポーネントのバージョン 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト

2.1.5

次の表に、このコンポーネントのバージョン 2.1.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト

2.1.4

次の表に、このコンポーネントのバージョン 2.1.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト

2.1.0

次の表に、このコンポーネントのバージョン 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト

構成

このコンポーネントに設定パラメータはありません。

ローカルログファイル

このコンポーネントはログを出力しません。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.11	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。

バージョン	変更
2.1.10	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.9	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.8	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.7	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.6	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.1.5	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.4	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.1.3	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.1.1	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.1.0	当初のバージョン

TensorFlow Lite ランタイム

TensorFlow Lite ランタイムコンポーネント (`variant.TensorFlowLite`) には、デバイスの仮想環境に TensorFlow Lite バージョン 2.5.0 とその依存関係をインストールするスクリプトが含まれています。TensorFlow Lite イメージ分類コンポーネントと TensorFlow Lite オブジェクト検出コンポー

ネントは、このランタイムコンポーネントを TensorFlow Lite をインストールするための依存関係として使用します。

> **Note**
>
> TensorFlow Lite ランタイムコンポーネント v2.5.6 以降では、 TensorFlow Lite ランタイムの既存のインストールとその依存関係が再インストールされます。この再インストールは、コアデバイスが互換性のあるバージョンの TensorFlow Lite とその依存関係を確実に実行するのに役立ちます。

別のランタイムを使用するには、このコンポーネントの recipe をテンプレートとして使用して、カスタム機械学習コンポーネントを作成することができます。

トピック

- バージョン
- タイプ
- オペレーティングシステム
- 要件
- 依存関係
- 構成
- 使用方法
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.5.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Amazon Linux 2 または Ubuntu 18.04 を実行している Greengrass コアデバイスの場合は、GNU C ライブラリ (glibc) バージョン 2.27 以降がデバイスにインストールされている必要があります。
- Raspberry Pi などの Armv7l デバイスでは、OpenCV-Python の依存関係がデバイスにインストールされています。次のコマンドを実行して依存関係をインストールします。

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

- Raspberry Pi OS Bullseye を実行する Raspberry Pi デバイスでは、次の要件を満たす必要があります。
  - NumPy 1.22.4 以降がデバイスにインストールされている。Raspberry Pi OS Bullseye には以前のバージョンの が含まれているため NumPy、次のコマンドを実行して NumPy デバイスでアップグレードできます。

    ```
    pip3 install --upgrade numpy
    ```

  - デバイスで、レガシーカメラスタックが有効になっていること。Raspberry Pi OS Bullseye には、デフォルトで新しいカメラスタックが含まれており有効化されていますが、これには互換性がないるため、レガシーカメラスタックを有効にしておく必要があります。

    レガシーカメラスタックを有効にするには

    1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

       ```
       sudo raspi-config
       ```

2. [Interface Options] (インターフェイスオプション) を選択します。
3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
4. Raspberry Pi を再起動します。

エンドポイントおよびポート

デフォルトでは、このコンポーネントは、インストーラスクリプトを使い、コアデバイスが使用するプラットフォームに応じて、`apt`、`yum`、`brew`、および `pip` コマンドを使用してパッケージをインストールします。このコンポーネントは、インストーラスクリプトを実行するために、さまざまなパッケージインデックスおよびリポジトリへのアウトバウンドリクエストを実行できる必要があります。このコンポーネントのアウトバウンドトラフィックがプロキシまたはファイアウォールを通過できるようにするには、コアデバイスがインストールに接続するパッケージインデックスとリポジトリのエンドポイントを特定する必要があります。

このコンポーネントのインストールスクリプトに必要なエンドポイントを特定するときは、次の点を考慮してください。

- エンドポイントは、コアデバイスのプラットフォームによって異なります。例えば、Ubuntu を実行するコアデバイスでは、`yum` または `brew` ではなく `apt` を使用します。さらに、同じパッケージインデックスを使用するデバイスは、異なるソースリストを持つ可能性があるため、異なるリポジトリからパッケージを取得する場合があります。
- 各デバイスにはパッケージの取得場所を定義する独自のソースリストがあるため、同じパッケージインデックスを使用する複数のデバイス間でエンドポイントが異なる場合があります。
- エンドポイントは時間の経過とともに変化する可能性があります。各パッケージインデックスは、パッケージをダウンロードするリポジトリの URL を提供し、パッケージの所有者は、パッケージインデックスが提供する URL を変更できます。

このコンポーネントがインストールする依存関係とインストーラスクリプトを無効にする方法の詳細については、「 UseInstaller設定パラメータ」を参照してください。

基本的な操作に必要なエンドポイントとポートの詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントと

その依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.5.14

次の表に、このコンポーネントのバージョン 2.5.14 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト

2.5.13

次の表に、このコンポーネントのバージョン 2.5.13 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト

2.5.12

次の表に、このコンポーネントのバージョン 2.5.12 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト

2.5.11

次の表に、このコンポーネントのバージョン 2.5.11 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト

2.5.10

次の表に、このコンポーネントのバージョン 2.5.10 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト

2.5.9

次の表に、このコンポーネントのバージョン 2.5.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト

2.5.8

次の表に、このコンポーネントのバージョン 2.5.8 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト

2.5.5 - 2.5.7

次の表に、このコンポーネントのバージョン 2.5.5 から 2.5.7 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト

2.5.3 and 2.5.4

次の表に、このコンポーネントのバージョン 2.5.3 および 2.5.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト

2.5.2

次の表に、このコンポーネントのバージョン 2.5.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト

2.5.1

次の表に、このコンポーネントのバージョン 2.5.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト

2.5.0

次の表に、このコンポーネントのバージョン 2.5.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

`MLRootPath`

(オプション) 推論コンポーネントがイメージを読み取り、推論結果を書き込む Linux コアデバイスのフォルダのパス。この値は、このコンポーネントを実行しているユーザーが読み取り/書き込みアクセス権を持つデバイスの任意の場所に変更できます。

デフォルト: *`/greengrass/v2`*`/work/variant.TensorFlowLite/greengrass_ml`

`WindowsMLRootPath`

この機能は、このコンポーネントの v1.6.6 以降で利用できます。

(オプション) 推論コンポーネントがイメージを読み取り、推論結果を書き込む Windows コアデバイスのフォルダのパス。この値は、このコンポーネントを実行しているユーザーが読み取り/書き込みアクセス権を持つデバイスの任意の場所に変更できます。

デフォルト: *`C:\greengrass\v2`*`\\work\\variant.DLR\\greengrass_ml`

`UseInstaller`

(オプション) このコンポーネントでインストーラスクリプトを使用して TensorFlow Lite とその依存関係をインストールするかどうかを定義する文字列値。サポートされている値は、`true` および `false` です。

TensorFlow Lite のインストールにカスタムスクリプト`false`を使用する場合、またはビルド済みの Linux イメージにランタイム依存関係を含める場合は、この値を に設定します。このコンポーネントを AWSが提供する TensorFlow Lite 推論コンポーネントで使用するには、依存関係を含む次のライブラリをインストールし、ML コンポーネント`ggc_user`を実行する などのシステムユーザーが使用できるようにします。

- Python 3.8 以降 (ご使用のバージョンの Python 用の `pip` を含む)。
- TensorFlow Lite v2.5.0
- NumPy
- OpenCV-Python
- AWS IoT Device SDK v2 for Python
- [AWS Common Runtime (CRT) Python] (　共通ランタイム (CRT) Python)
- Picamera (Raspberry Pi デバイス用)
- `awscam` モジュール (AWS DeepLens デバイス用)
- libGL (Linux デバイス用)

デフォルト: `true`

使用方法

`UseInstaller` 設定パラメータを に設定してこのコンポーネントを使用し`true`、デバイスに TensorFlow Lite とその依存関係をインストールします。コンポーネントは、 TensorFlow Lite に必要な OpenCV と NumPy ライブラリを含む仮想環境をデバイスに設定します。

> **Note**
>
> このコンポーネントのインストーラスクリプトは、デバイスの仮想環境を設定し、インストールされている機械学習フレームワークを使用するために必要な最新バージョンの追加システムライブラリもインストールします。これにより、デバイスで既存のシステムライブラリがアップグレードされる可能性があります。次の表で、サポートされている各オペレーティングシステムに、このコンポーネントがインストールするライブラリの一覧を確認してください。このインストールプロセスをカスタマイズする場合は、`UseInstaller` 設定パラメータを `false` に設定し、独自のインストーラスクリプトを開発します。

プラットフォーム	デバイスシステムにインストールされているライブラリ	仮想環境にインストールされているライブラリ
Armv7l	`build-essential` , `cmake`, `ca-certificates` , `git`	`setuptools` , `wheel`
Amazon Linux 2	`mesa-libGL`	なし
Ubuntu	`wget`	なし

推論コンポーネントをデプロイすると、このランタイムコンポーネントはまず、デバイスに既に TensorFlow Lite とその依存関係がインストールされているかどうかを確認します。そうでない場合は、ランタイムコンポーネントによってインストールされます。

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/variant.TensorFlowLite.log
```

Windows

```
C:\greengrass\v2\logs\variant.TensorFlowLite.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/variant.TensorFlowLite.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\variant.TensorFlowLite.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.5.14	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.5.13	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.5.12	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.5.11	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.5.10	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.5.9	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.5.8	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.5.7	バグ修正と機能向上 • `UseInstaller` インストールスクリプトを更新して libGL をインストールします。これは、一部の Linux プラットフォームではデフォルトで利用できません。 • このコンポーネントの仮想環境で常に Python 3.9 を使用するように、`UseInstaller` インストールスクリプトを更新します。この変更は、他のライブラリとの互換性を確保するのに役立ちます。
2.5.6	バグ修正と機能向上 • このコンポーネントを更新して TensorFlow Lite 2.5.0 (`tflite-runtime-2.5.0.post1`) の最新パッチをインストールし、Python 3.9 でこのコンポーネントを使用できるようにします。このコンポーネントがそのパッチのインストールに失敗すると、代わりに `tflite-runtime-2.5.0` をインストールします。 • このコンポーネントを更新して、 TensorFlow Lite の既存のインストールとその依存関係を再インストールします。この変更により、コアデバイスが互換性のあるバージョンの TensorFlow Lite とその依存関係を確実に実行できるようになります。
2.5.5	新機能 • Windows を実行するコアデバイスのサポートが追加されました。 • Windows コアデバイスで推論結果フォルダを設定するために使用できる新しい `WindowsMLRootPath` 設定パラメータを追加しました。
2.5.4	新機能 • このコンポーネントには、インストールスクリプトを無効にできる新しい `UseInstaller` 設定パラメータを追加されました。

バージョン	変更
2.5.3	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.5.2	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.5.1	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.5.0	当初のバージョン

Modbus-RTU プロトコルアダプタ

Modbus-RTU プロトコル アダプタ コンポーネント `aws.greengrass.Modbus` は、ローカル Modbus RTU デバイスからの情報をポーリングします。

このコンポーネントでローカル Modbus RTU デバイスから情報を要求するには、このコンポーネントがサブスクライブするトピックにメッセージをパブリッシュします。メッセージで、デバイスに送信する Modbus RTU 要求を指定します。次に、このコンポーネントは、 Modbus RTU 要求の結果を含むレスポンスをパブリッシュします。

> **Note**
>
> このコンポーネントは、 AWS IoT Greengrass V1 の Modbus RTU プロトコルアダプタコネクタと同様の機能を提供します。詳細については、「AWS IoT Greengrass V1 デベロッパーガイド」の「Modbus RTU プロトコールアダプタコネクタ」を参照してください。

トピック

- 構成
- 入力データ
- 出力データ
- Modbus RTU のリクエストとレスポンス
- ローカルログファイル
- ライセンス
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x
- 2.0.x

タイプ

このコンポーネントは Lambda コンポーネントです (`aws.greengrass.lambda`)。Greengrass nucleus は、Lambda ランチャーコンポーネントを使用してこのコンポーネントの Lambda 関数を実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、Linux コアデバイスにのみインストールできます。

要件

このコンポーネントには次の要件があります。

- コアデバイスは、Lambda 関数を実行するための要件を満たしている必要があります。コアデバイスが、コンテナ化された Lambda 関数を実行させる場合、そのデバイスは要件を満たす必要があります。詳細については、「Lambda 関数の要件」を参照してください。
- Python バージョン 3.7 がコアデバイスにインストールされ、PATH 環境変数に追加されています。

- AWS IoT Greengrass コアデバイスと Modbus デバイス間の物理接続。コアデバイスは、USB ポートなど、シリアルポートを介して Modbus RTU ネットワークに物理的に接続する必要があります。
- このコンポーネントから出力データを受信するには、このコンポーネントをデプロイするときに、次の設定更新プログラムをレガシーサブスクリプションルーターのコンポーネント (aws.greengrass.LegacySubscriptionRouter) のためにマージする必要があります。この設定は、このコンポーネントがレスポンスを公開するトピックを指定します。

  Legacy subscription router v2.1.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-modbus": {
        "id": "aws-greengrass-modbus",
        "source": "component:aws.greengrass.Modbus",
        "subject": "modbus/adapter/response",
        "target": "cloud"
      }
    }
  }
  ```

  Legacy subscription router v2.0.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-modbus": {
        "id": "aws-greengrass-modbus",
        "source": "arn:aws:lambda:region:aws:function:aws-greengrass-
  modbus:version",
        "subject": "modbus/adapter/response",
        "target": "cloud"
      }
    }
  }
  ```

  - *region* AWS リージョン は、使用する に置き換えます。
  - *#####*を、このコンポーネントが実行する Lambda 関数のバージョンに置き換えます。Lambda 関数のバージョンを確認するには、デプロイするこのコンポーネントのバージョンの recipe を確認する必要があります。AWS IoT Greengrass コンソールで、このコン

ポーネントの詳細ページを開き、[Lambda function] (Lambda 関数) の key-value ペアを見つけます。このキー値のペアには、Lambda 関数の名前とバージョンが含まれます。

> **Important**
>
> このコンポーネントをデプロイするたびに、レガシーサブスクリプションルーターの Lambda 関数のバージョンを更新する必要があります。これにより、デプロイするコンポーネントバージョンに正しい Lambda 関数のバージョンが使用されることが保証されます。

詳細については、「デプロイの作成」を参照してください。

- Modbus-RTU プロトコルアダプタは、VPC での実行がサポートされています。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.8

次の表に、このコンポーネントのバージョン 2.1.8 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.6

次の表に、このコンポーネントのバージョン 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.4 and 2.1.5

次の表に、このコンポーネントのバージョン 2.1.4 および 2.1.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト

依存関係	互換性のあるバージョン	依存関係タイプ
トークン交換サービス	^2.0.0	ハード

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.8 and 2.1.0

次の表に、このコンポーネントのバージョン 2.0.8 および 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.7

次の表に、このコンポーネントのバージョン 2.0.7 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.6

次の表に、このコンポーネントのバージョン 2.0.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.5

次の表に、このコンポーネントのバージョン 2.0.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.4

次の表に、このコンポーネントのバージョン 2.0.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト

依存関係	互換性のあるバージョン	依存関係タイプ
トークン交換サービス	^2.0.0	ハード

2.0.3

次の表に、このコンポーネントのバージョン 2.0.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.3 <2.1.0	ハード
Lambda ランチャー	>=1.0.0	ハード
Lambda ランタイム	>=1.0.0	ソフト
トークン交換サービス	>=1.0.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

> **Note**
>
> このコンポーネントのデフォルト設定には、Lambda 関数のパラメータが含まれます。デバイスにこのコンポーネントを設定するには、次のパラメータのみを編集することをお勧めします。

v2.1.x

`lambdaParams`

このコンポーネントの Lambda 関数のパラメータを含むオブジェクト。このオブジェクトには、次の情報が含まれます。

`EnvironmentVariables`

Lambda 関数のパラメータを含むオブジェクト。このオブジェクトには、次の情報が含まれます。

`ModbusLocalPort`

`/dev/ttyS2` など、コアデバイスの物理 Modbus シリアルポートへの絶対パス。

コンテナ内でこのコンポーネントを実行するには、コンポーネントがアクセスできるシステムデバイス (`containerParams.devices` 内) として、このパスを定義する必要があります。このコンポーネントは、デフォルトでコンテナ内で実行されます。

> **Note**
>
> このコンポーネントには、デバイスへの読み取り/書き込みアクセス権限が必要です。

`ModbusBaudRate`

(オプション) ローカル Modbus TCP デバイスとのシリアル通信におけるボーレートを指定する文字列値。

デフォルト: `9600`

`ModbusByteSize`

(オプション) ローカル Modbus TCP デバイスとのシリアル通信におけるバイトサイズを指定する文字列値。5、6、7、8 ビットのいずれかを選択します。

デフォルト: 8

`ModbusParity`

(オプション) ローカル Modbus TCP デバイスとのシリアル通信におけるデータの整合性を検証するために使用するパリティモード。

- `E` - 偶数のパリティでデータの整合性を検証します。
- `O` - 奇数のパリティでデータの整合性を検証します。
- `N` - データの整合性を検証しません。

デフォルト: `N`

`ModbusStopBits`

(オプション) ローカル Modbus TCP デバイスとのシリアル通信におけるバイトの終わりを示すビット数を指定する文字列値。

デフォルト: `1`

`containerMode`

(オプション) このコンポーネントのコンテナ化モード。次のオプションから選択します。

- `GreengrassContainer` - コンポーネントは、 AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。

  このオプションを指定した場合、コンテナに Modbus デバイスへのアクセスを許可するため、システムデバイス (`containerParams.devices` 内) を指定する必要があります。
- `NoContainer` - コンポーネントは、分離されたランタイム環境では実行されません。

デフォルト: `GreengrassContainer`

`containerParams`

(オプション) このコンポーネントのコンテナパラメータを含むオブジェクト。`containerMode` の `GreengrassContainer` を指定した場合、コンポーネントはこれらのパラメータを使用します。

このオブジェクトには、次の情報が含まれます。

`memorySize`

(オプション) コンポーネントに割り当てるメモリ量 (KB 単位)。

デフォルトは 512 MB (525,312 KB) です。

`devices`

(オプション) コンテナ内でコンポーネントがアクセスできるシステムデバイスを指定するオブジェクトです。

> **Important**
>
> コンテナ内でこのコンポーネントを実行するには、`ModbusLocalPort` 環境変数で設定するシステムデバイスを指定する必要があります。

このオブジェクトには、次の情報が含まれます。

`0` - これは文字列としての配列インデックスです。

次の情報が含まれるオブジェクト。

`path`

コアデバイスのシステムデバイスへのパス。これは、`ModbusLocalPort` に設定する値と同じであることが必要です。

`permission`

(オプション) コンテナからシステムデバイスにアクセスする許可。この値は `rw` である必要があり、コンポーネントにシステムデバイスへの読み取り/書き込みアクセス権限があることを指定します。

デフォルト: `rw`

`addGroupOwner`

(オプション) コンポーネントを実行するシステムグループを、システムデバイスの所有者として追加する有無。

デフォルト: `true`

`pubsubTopics`

(オプション) コンポーネントがメッセージを受信するためにサブスクライブするトピックを含むオブジェクト。各トピックと、コンポーネントが から MQTT トピックをサブスクライブするか、ローカルのパブリッシュ/サブスクライブトピックをサブスクライブ AWS IoT Core するかを指定できます。

このオブジェクトには、次の情報が含まれます。

`0` - これは文字列としての配列インデックスです。

次の情報が含まれるオブジェクト。

`type`

(オプション) このコンポーネントがメッセージをサブスクライブするために使用するパブリッシュ/サブスクライブメッセージングのタイプ。次のオプションから選択します。

- PUB_SUB - ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブします。このオプションを選択した場合、トピックに MQTT ワイルドカードを含める

ことはできません。このオプションを指定したときに、カスタムコンポーネントからメッセージを送信する方法の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。

- IOT_CORE – AWS IoT Core MQTT メッセージをサブスクライブします。このオプションを選択した場合、トピックに MQTT ワイルドカードを含めることができます。このオプションを指定したときに、カスタムコンポーネントからメッセージを送信する方法の詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。

デフォルト: PUB_SUB

`topic`

(オプション) コンポーネントがメッセージを受信するためにサブスクライブするトピック。type の IotCore を指定した場合、このトピックで MQTT ワイルドカード (+ および #) を使用できます。

Example 例: 設定マージの更新 (コンテナモード)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "ModbusLocalPort": "/dev/ttyS2"
    }
  },
  "containerMode": "GreengrassContainer",
  "containerParams": {
    "devices": {
      "0": {
        "path": "/dev/ttyS2",
        "permission": "rw",
        "addGroupOwner": true
      }
    }
  }
}
```

Example 例: 設定マージの更新 (コンテナモードなし)

```
{
  "lambdaExecutionParameters": {
```

```
    "EnvironmentVariables": {
      "ModbusLocalPort": "/dev/ttyS2"
    }
  },
  "containerMode": "NoContainer"
}
```

v2.0.x

`lambdaParams`

このコンポーネントの Lambda 関数のパラメータを含むオブジェクト。このオブジェクトには、次の情報が含まれます。

`EnvironmentVariables`

Lambda 関数のパラメータを含むオブジェクト。このオブジェクトには、次の情報が含まれます。

`ModbusLocalPort`

`/dev/ttyS2` など、コアデバイスの物理 Modbus シリアルポートへの絶対パス。

コンテナ内でこのコンポーネントを実行するには、コンポーネントがアクセスできるシステムデバイス (`containerParams.devices` 内) として、このパスを定義する必要があります。このコンポーネントは、デフォルトでコンテナ内で実行されます。

> **Note**
>
> このコンポーネントには、デバイスへの読み取り/書き込みアクセス権限が必要です。

`containerMode`

(オプション) このコンポーネントのコンテナ化モード。次のオプションから選択します。

- `GreengrassContainer` - コンポーネントは、 AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。

  このオプションを指定した場合、コンテナに Modbus デバイスへのアクセスを許可するため、システムデバイス (`containerParams.devices` 内) を指定する必要があります。
- `NoContainer` - コンポーネントは、分離されたランタイム環境では実行されません。

デフォルト: `GreengrassContainer`

`containerParams`

(オプション) このコンポーネントのコンテナパラメータを含むオブジェクト。`containerMode` の `GreengrassContainer` を指定した場合、コンポーネントはこれらのパラメータを使用します。

このオブジェクトには、次の情報が含まれます。

`memorySize`

(オプション) コンポーネントに割り当てるメモリ量 (KB 単位)。

デフォルトは 512 MB (525,312 KB) です。

`devices`

(オプション) コンテナ内でコンポーネントがアクセスできるシステムデバイスを指定するオブジェクトです。

> **Important**
> コンテナ内でこのコンポーネントを実行するには、`ModbusLocalPort` 環境変数で設定するシステムデバイスを指定する必要があります。

このオブジェクトには、次の情報が含まれます。

`0` - これは文字列としての配列インデックスです。

次の情報が含まれるオブジェクト。

`path`

コアデバイスのシステムデバイスへのパス。これは、`ModbusLocalPort` に設定する値と同じであることが必要です。

`permission`

(オプション) コンテナからシステムデバイスにアクセスする許可。この値は `rw` である必要があり、コンポーネントにシステムデバイスへの読み取り/書き込みアクセス権限があることを指定します。

デフォルト: `rw`

addGroupOwner

(オプション) コンポーネントを実行するシステムグループを、システムデバイスの所有者として追加する有無。

デフォルト: true

pubsubTopics

(オプション) コンポーネントがメッセージを受信するためにサブスクライブするトピックを含むオブジェクト。各トピックと、コンポーネントが から MQTT トピックをサブスクライブするか、ローカルのパブリッシュ/サブスクライブトピックをサブスクライブ AWS IoT Core するかを指定できます。

このオブジェクトには、次の情報が含まれます。

0 - これは文字列としての配列インデックスです。

次の情報が含まれるオブジェクト。

type

(オプション) このコンポーネントがメッセージをサブスクライブするために使用するパブリッシュ/サブスクライブメッセージングのタイプ。次のオプションから選択します。

- PUB_SUB - ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブします。このオプションを選択した場合、トピックに MQTT ワイルドカードを含めることはできません。このオプションを指定したときに、カスタムコンポーネントからメッセージを送信する方法の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。
- IOT_CORE – AWS IoT Core MQTT メッセージをサブスクライブします。このオプションを選択した場合、トピックに MQTT ワイルドカードを含めることができます。このオプションを指定したときに、カスタムコンポーネントからメッセージを送信する方法の詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。

デフォルト: PUB_SUB

topic

(オプション) コンポーネントがメッセージを受信するためにサブスクライブするトピック。type の IotCore を指定した場合、このトピックで MQTT ワイルドカード (+ および #) を使用できます。

Example 例: 設定マージの更新 (コンテナモード)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "ModbusLocalPort": "/dev/ttyS2"
    }
  },
  "containerMode": "GreengrassContainer",
  "containerParams": {
    "devices": {
      "0": {
        "path": "/dev/ttyS2",
        "permission": "rw",
        "addGroupOwner": true
      }
    }
  }
}
```

Example 例: 設定マージの更新 (コンテナモードなし)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "ModbusLocalPort": "/dev/ttyS2"
    }
  },
  "containerMode": "NoContainer"
}
```

入力データ

このコンポーネントは、次のトピックの Modbus RTU 要求パラメータを受入れて、Modbus RTU 要求をデバイスに送信します。デフォルトで、このコンポーネントはローカルのパブリッシュ/サブスクライブメッセージにサブスクライブします。カスタムコンポーネントからこのコンポーネントにメッセージをパブリッシュする方法の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。

デフォルトトピック (ローカルパブリッシュ/サブスクライブ): `modbus/adapter/request`

メッセージは、次のプロパティを受付けます。入力メッセージは JSON 形式である必要があります。

`request`

送信する Modbus RTU リクエストのパラメータ。

リクエストメッセージの形は、示す Modbus RTU 要求のタイプによって異なります。次のプロパティはすべての要求に必要です。

タイプ: 次の情報が含まれる `object`。

`operation`

実行するオペレーションの名前。例えば、Modbus RTU デバイスのコイルを読み取るため、`ReadCoilsRequest` を指定します。サポートされているオペレーションの詳細については、「Modbus RTU のリクエストとレスポンス」を参照してください。

タイプ: `string`

`device`

リクエストのターゲットデバイス。

この値は、`0` と `247` の間の整数である必要があります。

タイプ: `integer`

リクエストに含まれるその他のパラメータはオペレーションによって異なります。このコンポーネントは、ユーザーに代わって巡回冗長検査 (CRC) に対応して、データリクエストを検証します。

> **Note**
>
> リクエストに `address` プロパティが含まれる場合、その値を整数として指定する必要があります。例えば `"address": 1` です。

`id`

リクエストの任意の ID。このプロパティを使用して、入力リクエストを出力レスポンスにマッピングします。このプロパティを指定するとき、コンポーネントはこの値に対してレスポンスオブジェクトの `id` プロパティを設定します。

タイプ: `string`

Example 入力例: コイルの読み取りリクエスト

```
{
  "request": {
    "operation": "ReadCoilsRequest",
    "device": 1,
    "address": 1,
    "count": 1
  },
  "id": "MyRequest"
}
```

出力データ

このコンポーネントは、デフォルトで次の MQTT トピックに出力データとしてレスポンスを公開します。このトピックは、[legacy subscription router component] (レガシーサブスクリプションルーターコンポーネント) の設定で `subject` として指定する必要があります。カスタムコンポーネントでこのトピックに関するメッセージへサブスクライブする方法の詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。

デフォルトトピック (AWS IoT Core MQTT): `modbus/adapter/response`

レスポンスメッセージの形状は、リクエストオペレーションとレスポンスステータスによって異なります。例については、「リクエストとレスポンスの例」を参照してください。

レスポンスごとに、以下のプロパティが含まれます。

`response`

Modbus RTU デバイスからの応答。

タイプ: 次の情報が含まれる `object`。

`status`

リクエストのステータス。ステータスは、次のいずれかの値になります。

- `Success` - リクエストは有効で、コンポーネントはリクエストを Modbus RTU ネットワークに送信し、Modbus RTU ネットワークがレスポンスを返しました。

- `Exception` - リクエストは有効で、コンポーネントはリクエストを Modbus RTU ネットワークに送信し、Modbus RTU ネットワークが例外を返しました。詳細については、「レスポンスステータス: 例外」を参照してください。
- `No Response` – リクエストは無効で、コンポーネントが要求を Modbus RTU ネットワークに送信される前にエラーを検出しました。詳細については、「レスポンスステータス: No Response」を参照してください。

`operation`

コンポーネントが要求した操作。

`device`

コンポーネントが要求を送信したデバイス。

`payload`

Modbus RTU デバイスからの応答。`status` が `No Response` の場合、このオブジェクトはエラー (例えば、`[Input/Output] No Response received from the remote unit`) の説明を含む `error` プロパティのみが含まれています。

`id`

どのレスポンスがどのリクエストに対応しているか識別するために使用できるリクエストの ID。

> **Note**
>
> 書き込みオペレーションのレスポンスはリクエストのエコーのみです。書き込みレスポンスに有意な情報を含みませんが、要求が正常に処理されたか、失敗したか確認するため、レスポンスのステータスを確認することをお勧めします。

Example 出力例: 成功

```
{
  "response" : {
    "status" : "success",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "function_code": 1,
      "bits": [1]
```

```
    }
  },
  "id" : "MyRequest"
}
```

Example 出力例: 失敗

```
{
  "response" : {
    "status" : "fail",
    "error_message": "Internal Error",
    "error": "Exception",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "function_code": 129,
      "exception_code": 2
    }
  },
  "id" : "MyRequest"
}
```

その他の例については、「リクエストとレスポンスの例」を参照してください。

Modbus RTU のリクエストとレスポンス

このコネクタは、Modbus RTU のリクエストパラメータを入力データとして受け取り、レスポンスを出力データとして発行します。

以下の一般的なオペレーションがサポートされています。

リクエストのオペレーション名	レスポンスの関数コード
ReadCoilsRequest	01
ReadDiscreteInputsRequest	02
ReadHoldingRegistersRequest	03
ReadInputRegistersRequest	04
WriteSingleCoilRequest	05

リクエストのオペレーション名	レスポンスの関数コード
WriteSingleRegisterRequest	06
WriteMultipleCoilsRequest	15
WriteMultipleRegistersRequest	16
MaskWriteRegisterRequest	22
ReadWriteMultipleRegistersRequest	23

リクエストとレスポンスの例

以下に示しているのは、サポートされているオペレーションのリクエストとレスポンスの例です。

コイルの読み取り

リクエストの例:

```
{
  "request": {
    "operation": "ReadCoilsRequest",
    "device": 1,
    "address": 1,
    "count": 1
  },
  "id": "TestRequest"
}
```

レスポンスの例:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "function_code": 1,
      "bits": [1]
    }
  },
```

```
  "id" : "TestRequest"
}
```

個別入力の読み取り

リクエストの例:

```
{
  "request": {
    "operation": "ReadDiscreteInputsRequest",
    "device": 1,
    "address": 1,
    "count": 1
  },
  "id": "TestRequest"
}
```

レスポンスの例:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "ReadDiscreteInputsRequest",
    "payload": {
      "function_code": 2,
      "bits": [1]
    }
  },
  "id" : "TestRequest"
}
```

保持レジスタの読み取り

リクエストの例:

```
{
  "request": {
    "operation": "ReadHoldingRegistersRequest",
    "device": 1,
    "address": 1,
    "count": 1
  },
```

```
  "id": "TestRequest"
}
```

レスポンスの例:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "ReadHoldingRegistersRequest",
    "payload": {
      "function_code": 3,
      "registers": [20,30]
    }
  },
  "id" : "TestRequest"
}
```

入力レジスタの読み取り

リクエストの例:

```
{
  "request": {
    "operation": "ReadInputRegistersRequest",
    "device": 1,
    "address": 1,
    "count": 1
  },
  "id": "TestRequest"
}
```

単一コイルの書き込み

リクエストの例:

```
{
  "request": {
    "operation": "WriteSingleCoilRequest",
    "device": 1,
    "address": 1,
    "value": 1
  },
```

```
  "id": "TestRequest"
}
```

レスポンスの例:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "WriteSingleCoilRequest",
    "payload": {
      "function_code": 5,
      "address": 1,
      "value": true
    }
  },
  "id" : "TestRequest"
}
```

単一レジスタの書き込み

リクエストの例:

```
{
  "request": {
    "operation": "WriteSingleRegisterRequest",
    "device": 1,
    "address": 1,
    "value": 1
  },
  "id": "TestRequest"
}
```

複数コイルの書き込み

リクエストの例:

```
{
  "request": {
    "operation": "WriteMultipleCoilsRequest",
    "device": 1,
    "address": 1,
    "values": [1,0,0,1]
```

```
  },
  "id": "TestRequest"
}
```

レスポンスの例:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "WriteMultipleCoilsRequest",
    "payload": {
      "function_code": 15,
      "address": 1,
      "count": 4
    }
  },
  "id" : "TestRequest"
}
```

複数レジスタの書き込み

リクエストの例:

```
{
  "request": {
    "operation": "WriteMultipleRegistersRequest",
    "device": 1,
    "address": 1,
    "values": [20,30,10]
  },
  "id": "TestRequest"
}
```

レスポンスの例:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "WriteMultipleRegistersRequest",
    "payload": {
      "function_code": 23,
```

```
      "address": 1,
      "count": 3
    }
  },
  "id" : "TestRequest"
}
```

書き込みレジスタのマスク

リクエストの例:

```
{
  "request": {
    "operation": "MaskWriteRegisterRequest",
    "device": 1,
    "address": 1,
    "and_mask": 175,
    "or_mask": 1
  },
  "id": "TestRequest"
}
```

レスポンスの例:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "MaskWriteRegisterRequest",
    "payload": {
      "function_code": 22,
      "and_mask": 0,
      "or_mask": 8
    }
  },
  "id" : "TestRequest"
}
```

複数レジスタの読み書き

リクエストの例:

```
{
```

```
  "request": {
    "operation": "ReadWriteMultipleRegistersRequest",
    "device": 1,
    "read_address": 1,
    "read_count": 2,
    "write_address": 3,
    "write_registers": [20,30,40]
  },
  "id": "TestRequest"
}
```

レスポンスの例:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "ReadWriteMultipleRegistersRequest",
    "payload": {
      "function_code": 23,
      "registers": [10,20,10,20]
    }
  },
  "id" : "TestRequest"
}
```

> **Note**
>
> 応答には、コンポーネントが読み取るレジスタが含まれます。

レスポンスステータス: 例外

リクエスト書式が有効で、リクエストが正常に完了していない場合、例外が発生している可能性があります。この場合、レスポンスには以下の情報が含まれています。

- `status` は `Exception` に設定されています。
- `function_code` がリクエストの関数コード + 128 に等しい。
- `exception_code` に例外コードが含まれている。詳細については、Modbus 例外コードを参照してください。

例:

```
{
  "response": {
    "status": "fail",
    "error_message": "Internal Error",
    "error": "Exception",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "function_code": 129,
      "exception_code": 2
    }
  },
  "id": "TestRequest"
}
```

レスポンスステータス: No Response

このコネクタは Modbus リクエストの検証チェックを実行します。例えば、無効な書式や不足しているフィールドがないかどうかを確認します。検証に失敗すると、コネクタはリクエストを送信しません。代わりに、以下の情報を含むレスポンスを返します。

- `status` は `No Response` に設定されています。
- `error` にはエラーの理由が含まれています。
- `error_message` にはエラーメッセージが含まれています。

例:

```
{
  "response": {
    "status": "fail",
    "error_message": "Invalid address field. Expected <type 'int'>, got <type 'str'>",
    "error": "No Response",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "error": "Invalid address field. Expected Expected <type 'int'>, got <type
 'str'>"
    }
  },
```

```
  "id": "TestRequest"
}
```

リクエストのターゲットデバイスが存在しない場合、または Modbus RTU ネットワークが機能していない場合、No Response 形式を使用する `ModbusIOException` が返されます。

```
{
  "response": {
    "status": "fail",
    "error_message": "[Input/Output] No Response received from the remote unit",
    "error": "No Response",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "error": "[Input/Output] No Response received from the remote unit"
    }
  },
  "id": "TestRequest"
}
```

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

```
/greengrass/v2/logs/aws.greengrass.Modbus.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。を AWS IoT Greengrass ルートフォルダへのパス*/greengrass/v2*に置き換えます。

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.Modbus.log
  ```

ライセンス

このコンポーネントには、次のサードパーティーソフトウェア/ライセンス品が含まれています。

- pymodbus/BSD ライセンス

- pyserial/BSD ライセンス

このコンポーネントは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされています。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.8	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.1.7	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.6	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.5	バグ修正と機能向上 • `ReadDiscreteInput` オペレーションに関する問題を修正しました。
2.1.4	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.3	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.1.1	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.0	新機能 • Modbus RTU デバイスとのシリアル通信を設定するために指定できる `ModbusBaudRate` 、`ModbusByteSize` 、`ModbusParity` 、`ModbusStopBits` オプションを追加します。

バージョン	変更
2.0.8	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.0.7	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.0.6	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.0.5	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.0.4	Greengrass nucleus バージョン 2.1.0 のリリース用にバージョンが更新されました。
2.0.3	当初のバージョン

MQTT ブリッジ

MQTT ブリッジコンポーネント (`aws.greengrass.clientdevices.mqtt.Bridge`) は、クライアントデバイス、ローカル Greengrass パブリッシュ/サブスクライブ、AWS IoT Core の間の MQTT メッセージをリレーします。このコンポーネントを使用し、カスタムコンポーネントのクライアントデバイスからの MQTT メッセージを処理して、クライアントデバイスを AWS クラウド で同期できます。

> **Note**
>
> クライアントデバイスは、Greengrass コアデバイスに接続し、処理するために MQTT メッセージとデータを送信するローカル IoT デバイスです。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

このコンポーネントを使用して、次のメッセージブローカー間でメッセージをリレーできます:

- ローカル MQTT - ローカル MQTT ブローカーは、クライアントデバイスとコアデバイスの間のメッセージを処理します。

- ローカルパブリッシュ/サブスクライブ - ローカル Greengrass メッセージブローカーは、コアデバイスのコンポーネント間のメッセージを処理します。Greengrass コンポーネントでこれらのメッセージとやり取りする方法の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。
- AWS IoT Core - AWS IoT Core MQTT ブローカーは、IoT デバイスと AWS クラウド 宛先の間のメッセージを処理します。Greengrass コンポーネントでこれらのメッセージとやり取りする方法の詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。

> **Note**
>
> MQTT ブリッジは、クライアントデバイスが QoS 0 を使用してローカル MQTT ブローカーをパブリッシュとサブスクライブする場合でも、QoS 1 を使用して AWS IoT Core にパブリッシュとサブスクライブします。その結果、ローカル MQTT ブローカーのクライアントデバイスから AWS IoT Core に MQTT メッセージをリレーするとき、さらにレイテンシーが観察されることがあります。コアデバイスにおける MQTT 設定の詳細については、「MQTT タイムアウトとキャッシュ設定を設定する」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.3.x
- 2.2.x

- 2.1.x
- 2.0.x

タイプ

このコンポーネントはプラグインコンポーネント (`aws.greengrass.plugin`) です。Greengrass nucleus は、nucleus と同じ Java バーチャルマシン (JVM) でこのコンポーネントを実行します。コアデバイスでこのコンポーネントのバージョンを変更するとき、nucleus が再起動します。

このコンポーネントは、Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- デフォルトポートの 8883 以外のポートを使用するようにコアデバイスの MQTT ブローカーコンポーネントを設定する場合、MQTT ブリッジ v2.1.0 以降を使用する必要があります。ブローカーが動作するポートに接続するように設定します。
- MQTT ブリッジコンポーネントは、VPC での実行がサポートされています。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定

義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.3.0

次の表に、このコンポーネントのバージョン 2.3.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.2.0 <2.5.0	ハード

2.2.5 and 2.2.6

次の表に、このコンポーネントのバージョン 2.2.5 および 2.2.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.2.0 <2.5.0	ハード

2.2.3 and 2.2.4

次の表に、このコンポーネントのバージョン 2.2.3 および 2.2.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.2.0 <2.4.0	ハード

2.2.0 – 2.2.2

次の表に、このコンポーネントのバージョン 2.2.0 から 2.2.2 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.2.0 <2.3.0	ハード

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.0.0 <2.2.0	ハード

2.0.0 to 2.1.0

次の表に、このコンポーネントのバージョン 2.0.0 から 2.1.0 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.0.0 <2.1.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

2.3.0

`mqttTopicMapping`

ブリッジするトピックのマッピング。このコンポーネントは、ソーストピックのメッセージへサブスクライブし、受信したメッセージを宛先トピックにパブリッシュします。各トピックマッピングは、トピック、ソースタイプ、宛先タイプを定義します。

このオブジェクトには、次の情報が含まれます。

`topicMappingNameKey`

このトピックマッピングの名前。*`topicMappingNameKey`* を、このトピックマッピングを識別するのに役立つ名前に置き換えます。

このオブジェクトには、次の情報が含まれます。

`topic`

ソースブローカーとターゲットブローカーの間をつなぐトピックまたはトピックフィルター。

+ および # の MQTT トピックワイルドカードを使用すると、トピックフィルターに一致するすべてのトピックでメッセージをリレーできます。詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT トピック」を参照してください。

> **Note**
>
> Pubsub ソースブローカーとともに MQTT トピックのワイルドカードを使用するには、Greengrass nucleus コンポーネントの v2.6.0 以降を使用する必要があります。

`targetTopicPrefix`

このコンポーネントがメッセージをリレーするときにターゲットトピックに追加するプレフィックス。

`source`

ソースメッセージブローカー。次のオプションから選択します。

- `LocalMqtt` - クライアントデバイスが通信するローカル MQTT ブローカー。
- `Pubsub` - ローカル Greengrass パブリッシュ/サブスクライブメッセージブローカー。
- `IotCore` - AWS IoT Core MQTT メッセージブローカー。

> **Note**
>
> MQTT ブリッジは、クライアントデバイスが QoS 0 を使用してローカル MQTT ブローカーをパブリッシュとサブスクライブする場合でも、QoS 1 を使用して AWS IoT Core にパブリッシュとサブスクライブします。その結果、ローカル MQTT ブローカーのクライアントデバイスから AWS IoT Core に MQTT メッセージをリレーするとき、さらにレイテンシーが観察されることがあります。コアデバイスにおける MQTT 設定の詳細については、「MQTT タイムアウトとキャッシュ設定を設定する」を参照してください。

source と target は異なる必要があります。

target

ターゲット メッセージ ブローカー。次のオプションから選択します。

- LocalMqtt - クライアントデバイスが通信するローカル MQTT ブローカー。
- Pubsub - ローカル Greengrass パブリッシュ/サブスクライブメッセージブローカー。
- IotCore - AWS IoT Core MQTT メッセージブローカー。

> **Note**
>
> MQTT ブリッジは、クライアントデバイスが QoS 0 を使用してローカル MQTT ブローカーをパブリッシュとサブスクライブする場合でも、QoS 1 を使用して AWS IoT Core にパブリッシュとサブスクライブします。その結果、ローカル MQTT ブローカーのクライアントデバイスから AWS IoT Core に MQTT メッセージをリレーするとき、さらにレイテンシーが観察されることがあります。コアデバイスにおける MQTT 設定の詳細については、「MQTT タイムアウトとキャッシュ設定を設定する」を参照してください。

source と target は異なる必要があります。

mqtt5RouteOptions

(オプション) メッセージをソーストピックから宛先トピックにブリッジするためのトピックマッピングを設定するオプションを提供します。

このオブジェクトには、次の情報が含まれます。

mqtt5RouteOptionsNameKey

トピックマッピングのルートオプションの名前。*mqtt5RouteOptionsNameKey* を、mqttTopicMappingフィールドで定義されている一致する*topicMappingName##*に置き換えます。

このオブジェクトには、次の情報が含まれます。

noLocal

(オプション) 有効にすると、ブリッジはブリッジ自体がパブリッシュしたトピックに関するメッセージを転送しません。これを使うと、以下のようにループを防ぐことができます。

```
{
    "mqtt5RouteOptions": {
        "toIoTCore": {
            "noLocal": true
        }
    },
    "mqttTopicMapping": {
        "toIoTCore": {
            "topic": "device",
            "source": "LocalMqtt",
            "target": "IotCore"
        },
        "toLocal": {
            "topic": "device",
            "source": "IotCore",
            "target": "LocalMqtt"
        }
    }
}
```

`noLocal` は `source` が `LocalMqtt` であるルートでのみサポートされます。

デフォルト: false

retainAsPublished

(オプション) 有効にすると、ブリッジによって転送されるメッセージには、そのルートのブローカーにパブリッシュされたメッセージと同じ `retain` フラグが付きます。

`retainAsPublished` は `source` が `LocalMqtt` であるルートでのみサポートされます。

デフォルト: false

mqtt

(オプション) ローカルブローカーと通信するための MQTT プロトコル設定。

version

(オプション) ブリッジがローカルブローカーとの通信に使用する MQTT プロトコルのバージョン。nucleus 設定で選択した MQTT バージョンと同じである必要があります。

次から選択します。

- `mqtt3`
- `mqtt5`

`mqttTopicMapping` オブジェクトの `source` または `target` フィールドが `LocalMqtt` に設定されている場合は、MQTT ブローカーをデプロイする必要があります。`mqtt5` オプションを選択した場合は、[MQTT 5 ブローカー (EMQX)] を使用する必要があります。

デフォルト: `mqtt3`

ackTimeoutSeconds

(オプション) PUBACK、SUBACK、または UNSUBACK パケットが処理に失敗するまで待機する時間間隔。

デフォルト: 60

connAckTimeoutMs

(オプション) 接続をシャットダウンする前に CONNACK パケットを待機する時間間隔。

デフォルト: 20000 (20 秒)。

pingTimeoutMs

(オプション) ブリッジがローカルブローカーから PINGACK メッセージを受信するまで待機する時間 (ミリ秒)。待機時間がタイムアウトを超えると、ブリッジは MQTT 接続を閉じて再度開きます。この値は、`keepAliveTimeoutSeconds` より小さくなければなりません。

デフォルト: 30000 (30 秒)

keepAliveTimeout秒

(オプション) MQTT 接続を維持するためにブリッジが送信する各 PING メッセージを送信する間隔 (秒)。この値は `pingTimeoutMs` より大きくなければなりません。

デフォルト: 60

maxReconnectDelayMs

(オプション) MQTT が再接続するまでの最大時間 (秒) を指定します。

デフォルト: 30000 (30 秒)

minReconnectDelayMs

(オプション) MQTT が再接続するまでの最小時間 (秒) を指定します。

receiveMaximum

(オプション) ブリッジが送信できる未承認の QoS1 パケットの最大数。

デフォルト: 100

maximumPacketSize

クライアントが MQTT パケットで受け入れる最大バイト数。

デフォルト: null (制限なし)

sessionExpiryInterval

(オプション) ブリッジとローカルブローカー間のセッション継続を要求できる時間 (秒)。

デフォルト: 4294967295 (セッション有効期限なし)

`brokerUri`

(オプション) ローカル MQTT ブローカーの URI。デフォルトポート 8883 とは異なるポートを使用するように MQTT ブローカーを設定する場合、このパラメータを指定する必要があります。以下の形式を利用して、*###*を MQTT ブローカーが動作するポートに置き換えます: `ssl://localhost:`*`port`*。

デフォルト: `ssl://localhost:8883`

startupTimeoutSeconds

(オプション) コンポーネントが起動する最大時間 (秒)。このタイムアウトを超えている場合、コンポーネントの状態が `BROKEN` に変わります。

デフォルト: `120`

Example 例: 設定マージの更新

次の設定更新の例では、次の項目を指定します。

- `clients/+/hello/world` トピックフィルターに一致するトピックで、メッセージをクライアントデバイスから AWS IoT Core にリレーします。
- `clients/+/detections` トピックフィルターに一致するトピックで、メッセージをクライアントデバイスからローカルのパブリッシュ/サブスクライブにリレーし、ターゲットトピックに

events/input/ のプレフィックスを追加します。結果のターゲットトピックは、events/input/clients/+/detections トピックフィルターに一致します。

- clients/+/status トピックフィルターに一致するトピックで、メッセージをクライアントデバイスから AWS IoT Core にリレーし、ターゲットトピックに $aws/rules/StatusUpdateRule/ のプレフィックスを追加します。この例では、これらのメッセージを直接 StatusUpdateRule という名前の AWS IoT ルール にリレーして、Basic Ingest を使用したコスト削減を行います。

```
{
  "mqttTopicMapping": {
    "ClientDeviceHelloWorld": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
    "ClientDeviceEvents": {
      "topic": "clients/+/detections",
      "targetTopicPrefix": "events/input/",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ClientDeviceCloudStatusUpdate": {
      "topic": "clients/+/status",
      "targetTopicPrefix": "$aws/rules/StatusUpdateRule/",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

Example 例: MQTT 5 の設定

次の設定の例では、次の項目を更新します。

- ブリッジがローカルブローカーで MQTT 5 プロトコルを使用できるようにします。
- ClientDeviceHelloWorld トピックマッピングのパブリッシュされた設定として MQTT 保持を設定します。

```
{
```

```
  "mqttTopicMapping": {
    "ClientDeviceHelloWorld": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  },
  "mqtt5RouteOptions": {
    "ClientDeviceHelloWorld": {
      "retainAsPublished": true
    }
  },
  "mqtt": {
    "version": "mqtt5"
  }
}
```

2.2.6

`mqttTopicMapping`

ブリッジするトピックのマッピング。このコンポーネントは、ソーストピックのメッセージへサブスクライブし、受信したメッセージを宛先トピックにパブリッシュします。各トピックマッピングは、トピック、ソースタイプ、宛先タイプを定義します。

このオブジェクトには、次の情報が含まれます。

topicMappingNameKey

このトピックマッピングの名前。*topicMappingNameKey* を、このトピックマッピングを識別するのに役立つ名前に置き換えます。

このオブジェクトには、次の情報が含まれます。

`topic`

ソースブローカーとターゲットブローカーの間をつなぐトピックまたはトピックフィルター。

+ および # の MQTT トピックワイルドカードを使用すると、トピックフィルターに一致するすべてのトピックでメッセージをリレーできます。詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT トピック」を参照してください。

> **Note**
>
> Pubsub ソースブローカーとともに MQTT トピックのワイルドカードを使用するには、Greengrass nucleus コンポーネントの v2.6.0 以降を使用する必要があります。

`targetTopicPrefix`

このコンポーネントがメッセージをリレーするときにターゲットトピックに追加するプレフィックス。

`source`

ソースメッセージブローカー。次のオプションから選択します。

- `LocalMqtt` - クライアントデバイスが通信するローカル MQTT ブローカー。
- `Pubsub` - ローカル Greengrass パブリッシュ/サブスクライブメッセージブローカー。
- `IotCore` - AWS IoT Core MQTT メッセージブローカー。

> **Note**
>
> MQTT ブリッジは、クライアントデバイスが QoS 0 を使用してローカル MQTT ブローカーをパブリッシュとサブスクライブする場合でも、QoS 1 を使用して AWS IoT Core にパブリッシュとサブスクライブします。その結果、ローカル MQTT ブローカーのクライアントデバイスから AWS IoT Core に MQTT メッセージをリレーするとき、さらにレイテンシーが観察されることがあります。コアデバイスにおける MQTT 設定の詳細については、「MQTT タイムアウトとキャッシュ設定を設定する」を参照してください。

`source` と `target` は異なる必要があります。

`target`

ターゲット メッセージ ブローカー。次のオプションから選択します。

- `LocalMqtt` - クライアントデバイスが通信するローカル MQTT ブローカー。
- `Pubsub` - ローカル Greengrass パブリッシュ/サブスクライブメッセージブローカー。

- `IotCore` - AWS IoT Core MQTT メッセージブローカー。

  > **Note**
  >
  > MQTT ブリッジは、クライアントデバイスが QoS 0 を使用してローカル MQTT ブローカーをパブリッシュとサブスクライブする場合でも、QoS 1 を使用して AWS IoT Core にパブリッシュとサブスクライブします。その結果、ローカル MQTT ブローカーのクライアントデバイスから AWS IoT Core に MQTT メッセージをリレーするとき、さらにレイテンシーが観察されることがあります。コアデバイスにおける MQTT 設定の詳細については、「MQTT タイムアウトとキャッシュ設定を設定する」を参照してください。

  `source` と `target` は異なる必要があります。

`brokerUri`

(オプション) ローカル MQTT ブローカーの URI。デフォルトポート 8883 とは異なるポートを使用するように MQTT ブローカーを設定する場合、このパラメータを指定する必要があります。以下の形式を利用して、*###*を MQTT ブローカーが動作するポートに置き換えます: `ssl://localhost:`*`port`*。

デフォルト: `ssl://localhost:8883`

`startupTimeoutSeconds`

(オプション) コンポーネントが起動する最大時間 (秒)。このタイムアウトを超えている場合、コンポーネントの状態が `BROKEN` に変わります。

デフォルト: `120`

Example 例: 設定マージの更新

次の設定更新の例では、次の項目を指定します。

- `clients/+/hello/world` トピックフィルターに一致するトピックで、メッセージをクライアントデバイスから AWS IoT Core にリレーします。
- `clients/+/detections` トピックフィルターに一致するトピックで、メッセージをクライアントデバイスからローカルのパブリッシュ/サブスクライブにリレーし、ターゲットトピックに `events/input/` のプレフィックスを追加します。結果のターゲットトピックは、`events/input/clients/+/detections` トピックフィルターに一致します。

- clients/+/status トピックフィルターに一致するトピックで、メッセージをクライアントデバイスから AWS IoT Core にリレーし、ターゲットトピックに $aws/rules/StatusUpdateRule/ のプレフィックスを追加します。この例では、これらのメッセージを直接 StatusUpdateRule という名前の AWS IoT ルール にリレーして、Basic Ingest を使用したコスト削減を行います。

```
{
  "mqttTopicMapping": {
    "ClientDeviceHelloWorld": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
    "ClientDeviceEvents": {
      "topic": "clients/+/detections",
      "targetTopicPrefix": "events/input/",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ClientDeviceCloudStatusUpdate": {
      "topic": "clients/+/status",
      "targetTopicPrefix": "$aws/rules/StatusUpdateRule/",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

2.2.0 - 2.2.5

mqttTopicMapping

ブリッジするトピックのマッピング。このコンポーネントは、ソーストピックのメッセージへサブスクライブし、受信したメッセージを宛先トピックにパブリッシュします。各トピックマッピングは、トピック、ソースタイプ、宛先タイプを定義します。

このオブジェクトには、次の情報が含まれます。

topicMappingNameKey

このトピックマッピングの名前。*topicMappingNameKey* を、このトピックマッピングを識別するのに役立つ名前に置き換えます。

このオブジェクトには、次の情報が含まれます。

`topic`

ソースブローカーとターゲットブローカーの間をつなぐトピックまたはトピックフィルター。

+ および # の MQTT トピックワイルドカードを使用すると、トピックフィルターに一致するすべてのトピックでメッセージをリレーできます。詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT トピック」を参照してください。

> **Note**
>
> Pubsub ソースブローカーとともに MQTT トピックのワイルドカードを使用するには、Greengrass nucleus コンポーネントの v2.6.0 以降を使用する必要があります。

`targetTopicPrefix`

このコンポーネントがメッセージをリレーするときにターゲットトピックに追加するプレフィックス。

`source`

ソースメッセージブローカー。次のオプションから選択します。

- `LocalMqtt` - クライアントデバイスが通信するローカル MQTT ブローカー。
- `Pubsub` - ローカル Greengrass パブリッシュ/サブスクライブメッセージブローカー。
- `IotCore` - AWS IoT Core MQTT メッセージブローカー。

> **Note**
>
> MQTT ブリッジは、クライアントデバイスが QoS 0 を使用してローカル MQTT ブローカーをパブリッシュとサブスクライブする場合でも、QoS 1 を使用して AWS IoT Core にパブリッシュとサブスクライブします。その結果、ローカル MQTT ブローカーのクライアントデバイスから AWS IoT Core に MQTT メッセージをリレーするとき、さらにレイテンシーが観察されることがあります。コアデバイスにおける MQTT 設定の詳細については、「MQTT タイムアウトとキャッシュ設定を設定する」を参照してください。

source と target は異なる必要があります。

target

ターゲット メッセージ ブローカー。次のオプションから選択します。

- LocalMqtt - クライアントデバイスが通信するローカル MQTT ブローカー。
- Pubsub - ローカル Greengrass パブリッシュ/サブスクライブメッセージブローカー。
- IotCore - AWS IoT Core MQTT メッセージブローカー。

> **Note**
>
> MQTT ブリッジは、クライアントデバイスが QoS 0 を使用してローカル MQTT ブローカーをパブリッシュとサブスクライブする場合でも、QoS 1 を使用して AWS IoT Core にパブリッシュとサブスクライブします。その結果、ローカル MQTT ブローカーのクライアントデバイスから AWS IoT Core に MQTT メッセージをリレーするとき、さらにレイテンシーが観察されることがあります。コアデバイスにおける MQTT 設定の詳細については、「MQTT タイムアウトとキャッシュ設定を設定する」を参照してください。

source と target は異なる必要があります。

brokerUri

(オプション) ローカル MQTT ブローカーの URI。デフォルトポート 8883 とは異なるポートを使用するように MQTT ブローカーを設定する場合、このパラメータを指定する必要があります。以下の形式を利用して、*###*を MQTT ブローカーが動作するポートに置き換えます: ssl://localhost:*port*。

デフォルト: ssl://localhost:8883

Example 例: 設定マージの更新

次の設定更新の例では、次の項目を指定します。

- clients/+/hello/world トピックフィルターに一致するトピックで、メッセージをクライアントデバイスから AWS IoT Core にリレーします。
- clients/+/detections トピックフィルターに一致するトピックで、メッセージをクライアントデバイスからローカルのパブリッシュ/サブスクライブにリレーし、ターゲットトピックに

events/input/ のプレフィックスを追加します。結果のターゲットトピックは、events/input/clients/+/detections トピックフィルターに一致します。

- clients/+/status トピックフィルターに一致するトピックで、メッセージをクライアントデバイスから AWS IoT Core にリレーし、ターゲットトピックに $aws/rules/StatusUpdateRule/ のプレフィックスを追加します。この例では、これらのメッセージを直接 StatusUpdateRule という名前の AWS IoT ルール にリレーして、Basic Ingest を使用したコスト削減を行います。

```
{
  "mqttTopicMapping": {
    "ClientDeviceHelloWorld": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
    "ClientDeviceEvents": {
      "topic": "clients/+/detections",
      "targetTopicPrefix": "events/input/",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ClientDeviceCloudStatusUpdate": {
      "topic": "clients/+/status",
      "targetTopicPrefix": "$aws/rules/StatusUpdateRule/",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

2.1.x

`mqttTopicMapping`

ブリッジするトピックのマッピング。このコンポーネントは、ソーストピックのメッセージへサブスクライブし、受信したメッセージを宛先トピックにパブリッシュします。各トピックマッピングは、トピック、ソースタイプ、宛先タイプを定義します。

このオブジェクトには、次の情報が含まれます。

topicMappingNameKey

このトピックマッピングの名前。*topicMappingNameKey* を、このトピックマッピングを識別するのに役立つ名前に置き換えます。

このオブジェクトには、次の情報が含まれます。

`topic`

ソースブローカーとターゲットブローカーの間をつなぐトピックまたはトピックフィルター。

`LocalMqtt` または `IotCore` ソースブローカーを指定する場合、+ と # MQTT トピックワイルドカードを使用して、トピックフィルターに一致するすべてのトピックでメッセージをリレーします。詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT トピック」を参照してください。

`source`

ソース メッセージ ブローカー。次のオプションから選択します。

- `LocalMqtt` - クライアントデバイスが通信するローカル MQTT ブローカー。
- `Pubsub` - ローカル Greengrass パブリッシュ/サブスクライブメッセージブローカー。
- `IotCore` - AWS IoT Core MQTT メッセージブローカー。

> **Note**
>
> MQTT ブリッジは、クライアントデバイスが QoS 0 を使用してローカル MQTT ブローカーをパブリッシュとサブスクライブする場合でも、QoS 1 を使用して AWS IoT Core にパブリッシュとサブスクライブします。その結果、ローカル MQTT ブローカーのクライアントデバイスから AWS IoT Core に MQTT メッセージをリレーするとき、さらにレイテンシーが観察されることがあります。コアデバイスにおける MQTT 設定の詳細については、「MQTT タイムアウトとキャッシュ設定を設定する」を参照してください。

`source` と `target` は異なる必要があります。

`target`

ターゲット メッセージ ブローカー。次のオプションから選択します。

- `LocalMqtt` - クライアントデバイスが通信するローカル MQTT ブローカー。
- `Pubsub` - ローカル Greengrass パブリッシュ/サブスクライブメッセージブローカー。
- `IotCore` - AWS IoT Core MQTT メッセージブローカー。

> **Note**
>
> MQTT ブリッジは、クライアントデバイスが QoS 0 を使用してローカル MQTT ブローカーをパブリッシュとサブスクライブする場合でも、QoS 1 を使用して AWS IoT Core にパブリッシュとサブスクライブします。その結果、ローカル MQTT ブローカーのクライアントデバイスから AWS IoT Core に MQTT メッセージをリレーするとき、さらにレイテンシーが観察されることがあります。コアデバイスにおける MQTT 設定の詳細については、「MQTT タイムアウトとキャッシュ設定を設定する」を参照してください。

`source` と `target` は異なる必要があります。

`brokerUri`

(オプション) ローカル MQTT ブローカーの URI。デフォルトポート 8883 とは異なるポートを使用するように MQTT ブローカーを設定する場合、このパラメータを指定する必要があります。以下の形式を利用して、*###*を MQTT ブローカーが動作するポートに置き換えます: `ssl://localhost:`*`port`*。

デフォルト: `ssl://localhost:8883`

Example 例: 設定マージの更新

次の設定更新の例では、`clients/MyClientDevice1/hello/world` と `clients/MyClientDevice2/hello/world` のトピックについて、クライアントデバイスから AWS IoT Core にリレーするように指定しています。

```
{
  "mqttTopicMapping": {
    "ClientDevice1HelloWorld": {
      "topic": "clients/MyClientDevice1/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
```

```
    "ClientDevice2HelloWorld": {
      "topic": "clients/MyClientDevice2/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

2.0.x

`mqttTopicMapping`

ブリッジするトピックのマッピング。このコンポーネントは、ソーストピックのメッセージへサブスクライブし、受信したメッセージを宛先トピックにパブリッシュします。各トピックマッピングは、トピック、ソースタイプ、宛先タイプを定義します。

このオブジェクトには、次の情報が含まれます。

`topicMappingNameKey`

このトピックマッピングの名前。*`topicMappingNameKey`* を、このトピックマッピングを識別するのに役立つ名前に置き換えます。

このオブジェクトには、次の情報が含まれます。

`topic`

ソースブローカーとターゲットブローカーの間をつなぐトピックまたはトピックフィルター。

`LocalMqtt` または `IotCore` ソースブローカーを指定する場合、+ と # MQTT トピックワイルドカードを使用して、トピックフィルターに一致するすべてのトピックでメッセージをリレーします。詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT トピック」を参照してください。

`source`

ソース メッセージ ブローカー。次のオプションから選択します。

- `LocalMqtt` - クライアントデバイスが通信するローカル MQTT ブローカー。
- `Pubsub` - ローカル Greengrass パブリッシュ/サブスクライブメッセージブローカー。
- `IotCore` - AWS IoT Core MQTT メッセージブローカー。

> **Note**
>
> MQTT ブリッジは、クライアントデバイスが QoS 0 を使用してローカル MQTT ブローカーをパブリッシュとサブスクライブする場合でも、QoS 1 を使用して AWS IoT Core にパブリッシュとサブスクライブします。その結果、ローカル MQTT ブローカーのクライアントデバイスから AWS IoT Core に MQTT メッセージをリレーするとき、さらにレイテンシーが観察されることがあります。コアデバイスにおける MQTT 設定の詳細については、「MQTT タイムアウトとキャッシュ設定を設定する」を参照してください。

`source` と `target` は異なる必要があります。

`target`

ターゲット メッセージ ブローカー。次のオプションから選択します。

- `LocalMqtt` - クライアントデバイスが通信するローカル MQTT ブローカー。
- `Pubsub` - ローカル Greengrass パブリッシュ/サブスクライブメッセージブローカー。
- `IotCore` - AWS IoT Core MQTT メッセージブローカー。

> **Note**
>
> MQTT ブリッジは、クライアントデバイスが QoS 0 を使用してローカル MQTT ブローカーをパブリッシュとサブスクライブする場合でも、QoS 1 を使用して AWS IoT Core にパブリッシュとサブスクライブします。その結果、ローカル MQTT ブローカーのクライアントデバイスから AWS IoT Core に MQTT メッセージをリレーするとき、さらにレイテンシーが観察されることがあります。コアデバイスにおける MQTT 設定の詳細については、「MQTT タイムアウトとキャッシュ設定を設定する」を参照してください。

`source` と `target` は異なる必要があります。

Example 例: 設定マージの更新

次の設定更新の例では、`clients/MyClientDevice1/hello/world` と `clients/MyClientDevice2/hello/world` のトピックについて、クライアントデバイスから AWS IoT Core にリレーするように指定しています。

```
{
  "mqttTopicMapping": {
    "ClientDevice1HelloWorld": {
      "topic": "clients/MyClientDevice1/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
    "ClientDevice2HelloWorld": {
      "topic": "clients/MyClientDevice2/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

Linux

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.3.1	バグ修正と機能向上 ローカル MQTT クライアントが切断ループに入る問題を修正しました。
2.3.0	新機能 AWS IoT Core とローカル MQTT ソース間のブリッジ用の MQTT5 サポートを追加します。
2.2.6	新機能 新しい `startupTimeoutSeconds` 構成オプションを追加します。
2.2.5	クライアントデバイス認証バージョン 2.4.0 リリース用に、バージョンが更新されました。
2.2.4	Greengrass クライアントデバイス認証バージョン 2.3.0 リリース用に、バージョンが更新されました。
2.2.3	このバージョンには、バグ修正と機能向上が含まれています。
2.2.2	バグ修正と機能向上 • ロギング調整値。

バージョン	変更
2.2.1	バグ修正と機能向上 MQTT ブリッジによる MQTT トピックのサブスクライブの失敗を引き起こし得る問題を修正しました。
2.2.0	新機能 • ローカルの公開/サブスクライブをソースメッセージブローカーとして指定する場合の、MQTT トピックのワイルドカード (# および +) に関するサポートが追加されています。 この機能を使用するには、Greengrass nucleus コンポーネントの v2.6.0 以降が必要です。 • ターゲットトピックがメッセージを伝達するためのプレフィックスを追加するように MQTT ブリッジを設定するための、`targetTopicPrefix` オプションが追加されています。
2.1.1	バグ修正と機能向上 • このコンポーネントが設定リセット更新を処理する際に関する問題を修正しました。 • 証明書のローテーション時に MQTT クライアントが切断する頻度を減らします。
2.1.0	新機能 • デフォルト以外の MQTT ブローカーポートを使用できるようにする `brokerUri` パラメータを追加します。
2.0.1	このバージョンには、バグ修正と改善が含まれています。
2.0.0	当初のバージョン

MQTT 3.1.1 ブローカー (モケット)

モケット MQTT ブローカーコンポーネント (`aws.greengrass.clientdevices.mqtt.Moquette`) は、クライアントデバイスと Greengrass コアデバイス間の MQTT メッセージを処理します。このコンポーネントは、モケット MQTT ブロー

カーの修正バージョンを提供します。この MQTT ブローカーをデプロイして、軽量 MQTT ブローカーを実行します。MQTT ブローカーの選択方法の詳細については、「MQTT ブローカーを選択する」を参照してください。

このブローカーは、MQTT 3.1.1 プロトコルを実装します。これには、QoS 0、QoS 1、QoS 2 が保持するメッセージ、最終意志メッセージ、永続セッションに対するサポートが含まれます。

> **Note**
>
> クライアントデバイスは、Greengrass コアデバイスに接続し、処理するために MQTT メッセージとデータを送信するローカル IoT デバイスです。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

タイプ

このコンポーネントはプラグインコンポーネント (`aws.greengrass.plugin`) です。Greengrass nucleus は、nucleus と同じ Java バーチャルマシン (JVM) でこのコンポーネントを実行します。コアデバイスでこのコンポーネントのバージョンを変更するとき、nucleus が再起動します。

このコンポーネントは、Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- コアデバイスは、MQTT ブローカーが動作するポートで接続を受け入れられる必要があります。このコンポーネントは、デフォルトでポート 8883 で MQTT ブローカーを実行します。このコンポーネントを設定するとき、別のポートを指定できます。

  別のポートを指定し、MQTT ブリッジコンポーネントを使用して MQTT メッセージを他のブローカーにリレーする場合、MQTT ブリッジ v2.1.0 以降を使用する必要があります。MQTT ブローカーが動作するポートを使用するように設定します。

  別のポートを指定し、IP ディテクターコンポーネントを使用して MQTT ブローカーエンドポイントを管理する場合、IP ディテクタ v2.1.0 以降を使用する必要があります。MQTT ブローカーが動作するポートを報告するように設定します。
- Moquette MQTT ブローカーコンポーネントは、VPC での実行がサポートされています。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.3.2 – 2.3.5

次の表に、このコンポーネントのバージョン 2.3.2 から 2.3.5 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.2.0 <2.5.0	ハード

2.3.0 and 2.3.1

次の表に、このコンポーネントのバージョン 2.3.0 および 2.3.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.2.0 <2.4.0	ハード

2.2.0

次の表に、このコンポーネントのバージョン 2.2.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.2.0 <2.3.0	ハード

2.1.0

次の表に、このコンポーネントのバージョン 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.0.0 <2.2.0	ハード

2.0.0 - 2.0.2

次の表に、このコンポーネントのバージョン 2.0.0 から 2.0.2 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.0.0 <2.1.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

`moquette`

(オプション) 使用するモケット MQTT ブローカー設定。このコンポーネントでモケット設定オプションのサブセットを設定できます。詳細については、モケット設定ファイルのインラインコメントを参照してください。

このオブジェクトには、次の情報が含まれます。

`ssl_port`

(オプション) MQTT ブローカーが動作するポート。

> **Note**
>
> 別のポートを指定し、MQTT ブリッジコンポーネントを使用して MQTT メッセージを他のブローカーにリレーする場合、MQTT ブリッジ v2.1.0 以降を使用する必要があります。MQTT ブローカーが動作するポートを使用するように設定します。

別のポートを指定し、IP ディテクターコンポーネントを使用して MQTT ブローカーエンドポイントを管理する場合、IP ディテクタ v2.1.0 以降を使用する必要があります。MQTT ブローカーが動作するポートを報告するように設定します。

デフォルト: 8883

`host`

(オプション) MQTT ブローカーがバインドするインターフェイス。例えば、MQTT ブローカーが特定のローカルネットワークにのみバインドするように、このパラメータを変更できます。

デフォルト: `0.0.0.0` (すべてのネットワークインターフェイスにバインドする)

startupTimeoutSeconds

(オプション) コンポーネントが起動する最大時間 (秒)。このタイムアウトを超えている場合、コンポーネントの状態が `BROKEN` に変わります。

デフォルト: `120`

Example 例: 設定マージの更新

次の設定例では、MQTT ブローカーをポート 443 で操作するように指定しています。

```
{
  "moquette": {
    "ssl_port": "443"
  }
}
```

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.3.5	バグ修正と機能向上 • Moquette をバージョン 0.17 に更新しました
2.3.4	バグ修正と機能向上 • クライアントのメッセージ送受信時に、クライアント ID が重複しているために無効なセッションエラーが発生することがある問題を修正しました。この問題により、クライアントのセッションが終了する事象が発生していました。
2.3.3	新機能 新しい `startupTimeoutSeconds` 構成オプションを追加します。

バージョン	変更
2.3.2	クライアントデバイス認証バージョン 2.4.0 リリース用に、バージョンが更新されました。
2.3.1	バグ修正と機能向上 • セッションが無効なため、再接続を試みた後にクライアントが切断される可能性がある競合状態を修正します。
2.3.0	証明書チェーンのサポートを追加しました。
2.2.0	クライアントデバイス認証バージョン 2.2.0 リリース用にバージョンが更新されました。
2.1.0	バグ修正と機能向上 • このコンポーネントがモケットバージョン 0.16 使用するように更新します。このバージョンは、パフォーマンスを向上させて、他にもいくつかの改善点が含まれています。 • 特定のシナリオで、ローカル MQTT サーバー証明書が意図した回数よりも頻繁にローテーションされる問題を修正しました。 この修正を適用するには、v2.1.0 以降のクライアントデバイス認証コンポーネントも使用する必要があります。
2.0.2	バグ修正と機能向上 • MQTT メッセージの最大サイズを 8,092 バイトから 128 キロバイトに増やします。メッセージサイズ上限にメッセージヘッダーが含まれているため、有効な MQTT メッセージペイロード上限はわずかに小さくなります。 • `ssl_port` パラメータの整数値にサポートを追加します。
2.0.1	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.0.0	当初のバージョン

MQTT 5 ブローカー (EMQX)

EMQX MQTT ブローカーコンポーネント (`aws.greengrass.clientdevices.mqtt.EMQX`) は、クライアントデバイスと Greengrass コアデバイス間の MQTT メッセージを処理します。このコンポーネントは、EMQX MQTT 5.0 ブローカーの修正バージョンを提供します。この MQTT ブローカーをデプロイして、クライアントデバイスとコアデバイスの間の通信に MQTT 5 機能を使用します。MQTT ブローカーの選択方法の詳細については、「MQTT ブローカーを選択する」を参照してください。

このブローカーは、MQTT 5.0 プロトコルを実装します。セッションとメッセージの有効期限、ユーザープロパティ、共有サブスクリプション、トピックエイリアスなどのサポートが含まれます。MQTT 5 は MQTT 3.1.1 と下位互換性があるので、モケット MQTT 3.1.1 ブローカーを実行する場合、EMQX MQTT 5 ブローカーに置き換えることができ、クライアントデバイスは通常どおり接続して動作し続けることができます。

> **Note**
>
> クライアントデバイスは、Greengrass コアデバイスに接続し、処理するために MQTT メッセージとデータを送信するローカル IoT デバイスです。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.0.x
- 1.2.x
- 1.1.x
- 1.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- コアデバイスは、MQTT ブローカーが動作するポートで接続を受け入れられる必要があります。このコンポーネントは、デフォルトでポート 8883 で MQTT ブローカーを実行します。このコンポーネントを設定するとき、別のポートを指定できます。

  別のポートを指定し、MQTT ブリッジコンポーネントを使用して MQTT メッセージを他のブローカーにリレーする場合、MQTT ブリッジ v2.1.0 以降を使用する必要があります。MQTT ブローカーが動作するポートを使用するように設定します。

  別のポートを指定し、IP ディテクターコンポーネントを使用して MQTT ブローカーエンドポイントを管理する場合、IP ディテクタ v2.1.0 以降を使用する必要があります。MQTT ブローカーが動作するポートを報告するように設定します。

- Linux コアデバイスでは、次のように Docker がコアデバイスにインストールおよび設定されています。
  - Docker Engine 1.9.1 以降が Greengrass コアにインストールされていいること。バージョン 20.10 は、AWS IoT Greengrass Core ソフトウェアとの動作が確認されている最新バージョンです。Docker コンテナを実行するコンポーネントをデプロイする前に、コアデバイスに直接、Docker をインストールしておく必要があります。
  - このコンポーネントをデプロイする前に、Docker デーモンがコアデバイス上で起動し、実行されています。
  - このコンポーネントを実行するシステムユーザーには、ルート権限または管理者権限が必要です。または、このコンポーネントを `docker` グループ内のシステムユーザーとして実行し、このコンポーネントの `requiresPrivileges` オプションを `false` に設定して特権なしで EQMX MQTT ブローカーを実行できます。
- EMQX MQTT ブローカーコンポーネントは、VPC での実行がサポートされています。
- EMQX MQTT ブローカーコンポーネントは、 `armv7`プラットフォームではサポートされていません。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.0.0

次の表に、このコンポーネントのバージョン 2.0.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.2.0 <2.5.0	ハード

1.2.2 – 1.2.3

次の表に、このコンポーネントのバージョン 1.2.2 から 1.2.3 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.2.0 <2.5.0	ハード

1.2.0 and 1.2.1

次の表に、このコンポーネントのバージョン 1.2.0 および 1.2.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.2.0 <2.4.0	ハード

1.0.0 and 1.1.0

次の表に、このコンポーネントのバージョン 1.0.0 および 1.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
クライアントデバイス認証	>=2.2.0 <2.3.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

2.0.0

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

> **Important**
>
> MQTT 5 ブローカー (EMQX) コンポーネントのバージョン 2 を使用している場合は、設定ファイルを更新する必要があります。バージョン 1 の設定ファイルはバージョン 2 では機能しません。

emqxConfig

(オプション) 使用する EMQX MQTT ブローカー設定。このコンポーネントで EMQX 設定オプションを設定できます。

EMQX ブローカーを使用する場合、Greengrass ではデフォルト設定が使用されます。このフィールドを使用して変更しない限り、この設定が使用されます。

以下の設定を変更すると、EMQX Broker コンポーネントが再起動します。他の設定変更は、コンポーネントを再起動せずに適用されます。

- `emqxConfig/cluster`
- `emqxConfig/node`
- `emqxConfig/rpc`

> **Note**
>
> `aws.greengrass.clientdevices.mqtt.EMQX` は、セキュリティに配慮したオプションを設定することを許可します。これらには、TLS 設定、認証、承認プロバイダーが含まれます。相互 TLS 認証と Greengrass クライアントデバイス認証プロバイダーを使用するデフォルト設定を推奨します。

Example 例: デフォルト設定

次の例は、MQTT 5 (EMQX) ブローカーのデフォルト設定を示しています。これらの設定は、`emqxConfig` 設定を使用してオーバーライドできます。

```
{
  "authorization": {
    "no_match": "deny",
    "sources": []
```

```
  },
  "node": {
    "cookie": "<placeholder>"
  },
  "listeners": {
     "ssl": {
       "default": {
         "ssl_options": {
           "keyfile": "{work:path}\\data\\key.pem",
           "certfile": "{work:path}\\data\\cert.pem",
           "cacertfile": null,
           "verify": "verify_peer",
           "versions": ["tlsv1.3", "tlsv1.2"],
           "fail_if_no_peer_cert": true
         }
       }
     },
     "tcp": {
       "default": {
         "enabled": false
       }
     },
     "ws": {
       "default": {
         "enabled": false
       }
     },
     "wss": {
       "default": {
         "enabled": false
       }
     }
  },
  "plugins": {
    "states": [{"name_vsn": "gg-1.0.0", "enable": true}],
    "install_dir": "plugins"
  }
}
```

authMode

(オプション) ブローカーの認証プロバイダーを設定します。次のいずれかの値を指定できます。

- `enabled` – (デフォルト) Greengrass 認証および承認プロバイダーを使用します。
- `bypass_on_failure` – Greengrass 認証プロバイダーを使用し、Greengrass が認証または承認を拒否する場合は、EMQX プロバイダーチェーン内の残りの認証プロバイダーを使用します。
- `bypass` – Greengrass プロバイダーが無効になっています。認証と承認は EMQX プロバイダーチェーンによって処理されます。

`requiresPrivilege`

(オプション) Linux コアデバイスでは、ルート権限または管理者権限なしで EMQX MQTT ブローカーを実行するように指定できます。このオプションを `false` に設定した場合、このコンポーネントを実行するシステムユーザーは、`docker` グループのメンバーである必要があります。

デフォルト: `true`

`startupTimeoutSeconds`

(オプション) EMQX MQTT ブローカーが起動する最大時間 (秒)。このタイムアウトを超えている場合、コンポーネントの状態が `BROKEN` に変わります。

デフォルト: `90`

`ipcTimeoutSeconds`

(オプション) Greengrass nucleus がプロセス間通信 (IPC) リクエストに応答するまでコンポーネントが待機する最大時間 (秒)。このコンポーネントがクライアントデバイスが認証されているかどうかを確認するときにタイムアウトエラーを報告する場合は、この数値を増やします。

デフォルト: `5`

`crtLogLevel`

(オプション) AWS Common Runtime (CRT) ライブラリのログレベル。

デフォルトは EMQX MQTT ブローカーのログレベル (`emqx` の `log.level`)。

`restartIdentifier`

(オプション) EMQX MQTT ブローカーを再起動するには、このオプションを設定します。この設定値が変更されると、このコンポーネントは MQTT ブローカーを再起動します。このオプションを使用して、クライアントデバイスを強制的に切断できます。

`dockerOptions`

(オプション) Docker コマンドラインにパラメータを追加するには、Linux オペレーティングシステムでのみこのオプションを設定します。例えば、追加のポートをマッピングするには、-p Docker パラメータを使用します。

```
"-p 1883:1883"
```

Example 例: v1.x 設定ファイルを v2.x に更新する

次の例は、v1.x 設定ファイルをバージョン 2.x に更新するために必要な変更を示しています。

バージョン 1.x の設定ファイル:

```
{
    "emqx": {
        "listener.ssl.external": "443",
        "listener.ssl.external.max_connections": "1024000",
        "listener.ssl.external.max_conn_rate": "500",
        "listener.ssl.external.rate_limit": "50KB,5s",
        "listener.ssl.external.handshake_timeout": "15s",
        "log.level": "warning"
    },
    "mergeConfigurationFiles": {
        "etc/plugins/aws_greengrass_emqx_auth.conf": "auth_mode=enabled\n
 use_greengrass_managed_certificates=true\n"
    }
}
```

v2 と同等の設定ファイル:

```
{
    "emqxConfig": {
        "listeners": {
            "ssl": {
                "default": {
                   "bind": "8883",
                   "max_connections": "1024000",
                   "max_conn_rate": "500",
                   "handshake_timeout": "15s",
                }
```

```
            }
        },
        "log": {
            "console": {
              "enable": true,
              "level": "warning"
            }
        }
    },
    "authMode": "enabled"
}
```

listener.ssl.external.rate_limit 設定エントリに相当するものはありません。use_greengrass_managed_certificates 設定オプションは削除されました。

Example 例: ブローカーに新しいポートを設定する

次の例では、MQTT ブローカーが動作するポートをデフォルトの 8883 からポート 1234 に変更します。Linux を使用している場合は、dockerOptions フィールドを含めてください。

```
{
  "emqxConfig": {
    "listeners": {
      "ssl": {
        "default": {
          "bind": 1234
        }
      }
    }
  },
  "dockerOptions": "-p 1234:1234"
}
```

Example 例: MQTT ブローカーのログレベルの調整

次の例では、MQTT ブローカーのログレベルを debug に変更しています。次のログレベルから選択できます。

- debug
- info
- notice

- warning
- error
- critical
- alert
- emergency

デフォルトのログレベルは warning です。

```
{
  "emqxConfig": {
    "log": {
      "console": {
         "level": "debug"
      }
    }
  }
}
```

Example 例: EMQX ダッシュボードを有効にする

次の例では EMQX ダッシュボードを有効にして、ブローカーの監視と管理を行えるようにします。Linux を使用している場合は、dockerOptions フィールドを含めてください。

```
{
  "emqxConfig": {
    "dashboard": {
      "listeners": {
        "http": {
          "bind": 18083
        }
      }
    }
  },
  "dockerOptions": "-p 18083:18083"
}
```

1.0.0 - 1.2.2

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

`emqx`

(オプション) 使用する EMQX MQTT ブローカー設定。このコンポーネントで EMQX 設定オプションのサブセットを設定できます。

このオブジェクトには、次の情報が含まれます。

`listener.ssl.external`

(オプション) MQTT ブローカーが動作するポート。

> **Note**
>
> 別のポートを指定し、MQTT ブリッジコンポーネントを使用して MQTT メッセージを他のブローカーにリレーする場合、MQTT ブリッジ v2.1.0 以降を使用する必要があります。MQTT ブローカーが動作するポートを使用するように設定します。
> 別のポートを指定し、IP ディテクターコンポーネントを使用して MQTT ブローカーエンドポイントを管理する場合、IP ディテクタ v2.1.0 以降を使用する必要があります。MQTT ブローカーが動作するポートを報告するように設定します。

デフォルト: `8883`

`listener.ssl.external.max_connections`

(オプション) MQTT ブローカーがサポートする同時接続の最大数。

デフォルト: `1024000`

`listener.ssl.external.max_conn_rate`

(オプション) MQTT ブローカーが受信できる 1 秒あたりの新規接続の最大数。

デフォルト: `500`

`listener.ssl.external.rate_limit`

(オプション) MQTT ブローカーへのすべての接続の帯域幅制限。帯域幅とその帯域幅の期間をカンマ (`,`) で区切って `bandwidth,duration` の形式で指定します。例えば、`50KB,5s` と指定して、MQTT ブローカーを 5 秒ごとに 50 キロバイト (KB) のデータに制限できます。

`listener.ssl.external.handshake_timeout`

(オプション) MQTT ブローカーが新しい接続の認証が終了するまで待機する時間。

デフォルト: `15s`

`mqtt.max_packet_size`

(オプション) MQTT メッセージの最大サイズ。

デフォルト: 268435455 (256 MB マイナス 1)

`log.level`

(オプション) MQTT ブローカーのログレベル。次のオプションから選択します。

- `debug`
- `info`
- `notice`
- `warning`
- `error`
- `critical`
- `alert`
- `emergency`

デフォルトのログレベルは `warning` です。

`requiresPrivilege`

(オプション) Linux コアデバイスでは、ルート権限または管理者権限なしで EMQX MQTT ブローカーを実行するように指定できます。このオプションを `false` に設定した場合、このコンポーネントを実行するシステムユーザーは、`docker` グループのメンバーである必要があります。

デフォルト: `true`

`startupTimeoutSeconds`

(オプション) EMQX MQTT ブローカーが起動する最大時間 (秒)。このタイムアウトを超えている場合、コンポーネントの状態が `BROKEN` に変わります。

デフォルト: `90`

`ipcTimeoutSeconds`

(オプション) Greengrass nucleus がプロセス間通信 (IPC) リクエストに応答するまでコンポーネントが待機する最大時間 (秒)。このコンポーネントがクライアントデバイスが認証されているかどうかを確認するときにタイムアウトエラーを報告する場合は、この数値を増やします。

デフォルト: 5

`crtLogLevel`

(オプション) AWS Common Runtime (CRT) ライブラリのログレベル。

デフォルトは EMQX MQTT ブローカーのログレベル (`emqx` の `log.level`)。

`restartIdentifier`

(オプション) EMQX MQTT ブローカーを再起動するには、このオプションを設定します。この設定値が変更されると、このコンポーネントは MQTT ブローカーを再起動します。このオプションを使用して、クライアントデバイスを強制的に切断できます。

`dockerOptions`

(オプション) Docker コマンドラインにパラメータを追加するには、Linux オペレーティングシステムでのみこのオプションを設定します。例えば、追加のポートをマッピングするには、`-p` Docker パラメータを使用します。

```
"-p 1883:1883"
```

`mergeConfigurationFiles`

(オプション) 指定した EMQX 設定ファイルのデフォルトに追加したり、デフォルトをオーバーライドしたりするには、このオプションを設定します。設定ファイルとその形式については、EMQX 4.0 ドキュメントの「Configuration」(設定) を参照してください。指定した値は、設定ファイルに付加されます。

次の例では、`etc/emqx.conf` ファイルを更新します。

```
"mergeConfigurationFiles": {
    "etc/emqx.conf": "broker.sys_interval=30s\nbroker.sys_heartbeat=10s"
},
```

EMQX によってサポートされている設定ファイルに加えて、Greengrass は、`etc/plugins/aws_greengrass_emqx_auth.conf` と呼ばれる EMQX 用の Greengrass 認証プラグインを設定するファイルをサポートしています。サポートされているオプションは `auth_mode` と `use_greengrass_managed_certificates` の 2 つです。別の認証プロバイダーを使用するには、`auth_mode` オプションを次のいずれかに設定します。

- `enabled` – (デフォルト) Greengrass 認証および承認プロバイダーを使用します。
- `bypass_on_failure` – Greengrass 認証プロバイダーを使用し、Greengrass が認証または承認を拒否する場合は、EMQX プロバイダーチェーン内の残りの認証プロバイダーを使用します。
- `bypass` – Greengrass プロバイダーが無効になっています。その後、認証と承認は EMQX プロバイダーチェーンによって処理されます。

`use_greengrass_managed_certificates` が `true` の場合、このオプションは Greengrass がブローカー TLS 証明書を管理することを示します。`false` の場合は、別のソースを通じて証明書を提供していることを示しています。

次の例では、`etc/plugins/aws_greengrass_emqx_auth.conf` 設定ファイルのデフォルトを更新します。

```
"mergeConfigurationFiles": {
    "etc/plugins/aws_greengrass_emqx_auth.conf": "auth_mode=enabled\n
 use_greengrass_managed_certificates=true\n"
  },
```

> **Note**
>
> `aws.greengrass.clientdevices.mqtt.EMQX` は、セキュリティに配慮したオプションを設定することを許可します。これらには、TLS 設定、認証、承認プロバイダーが含まれます。推奨設定は、相互 TLS 認証と Greengrass Client Device Auth プロバイダーを使用するデフォルト設定です。

`replaceConfigurationFiles`

(オプション) 指定した EMQX 設定ファイルを置き換えるには、このオプションを設定します。指定した値は、既存の設定ファイル全体を置き換えます。このセクションでは `etc/emqx.conf` ファイルを指定できません。`mergeConfigurationFile` を使用して `etc/emqx.conf` を変更する必要があります。

Example 例: 設定マージの更新

次の設定例では、MQTT ブローカーをポート 443 で操作するように指定しています。

```
{
  "emqx": {
    "listener.ssl.external": "443",
    "listener.ssl.external.max_connections": "1024000",
    "listener.ssl.external.max_conn_rate": "500",
    "listener.ssl.external.rate_limit": "50KB,5s",
    "listener.ssl.external.handshake_timeout": "15s",
    "log.level": "warning"
  },
  "requiresPrivilege": "true",
  "startupTimeoutSeconds": "90",
  "ipcTimeoutSeconds": "5"
}
```

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/aws.greengrass.clientdevices.mqtt.EMQX.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.clientdevices.mqtt.EMQX.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.clientdevices.mqtt.EMQX.log
  ```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\aws.greengrass.clientdevices.mqtt.EMQX.log -
Tail 10 -Wait
```

ライセンス

Windows オペレーティングシステムでは、このソフトウェアには、マイクロソフト ソフトウェア ライセンス条項 - MICROSOFT VISUAL STUDIO COMMUNITY 2022 に基づいて配布されたコードが含まれています。このソフトウェアをダウンロードすると、このコードのライセンス条項に同意したことになります。

このコンポーネントは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされています。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

v2.x

バージョン	変更
2.0.0	MQTT 5 ブローカー (EMQX) のこのバージョンでは、バージョン 1.x とは異なる設定パラメータが必要です。バージョン 1.x でデフォルト以外の設定を使用している場合は、2.x のコンポーネント設定を更新する必要があります。詳細については、「構成」を参照してください。 新機能 • MQTT ブローカーを EMQX 5.1.1 にアップグレードしました。 • コンポーネントを再起動せずにブローカーの設定を変更できるようになりました。 更新 • `emqx`、`mergeConfigurationFiles` 、`replaceConfigurationFiles` 設定フィールドに代わる、 `emqxConfig` 設定フィールドが新たに加わりました。

v1.x

バージョン	変更
1.2.3	バグ修正と機能向上 • クライアントを切断して再認証することで以前に認証した後に、クライアントが EMQX と対話できなくなる問題を修正しました。
1.2.2	クライアントデバイス認証バージョン 2.4.0 リリース用に、バージョンが更新されました。
1.2.1	バグ修正と機能向上 • Visual C++ 再配布可能パッケージがまだ存在しない場合に Windows 上でコンポーネントが起動しない問題を修正します。 • EMQX をバージョン 4.4.14 に更新します。
1.2.0	証明書チェーンのサポートを追加しました。
1.1.0	新機能 • ブローカーオプションやプラグインを含む EMQX 設定のサポートを追加します。 バグ修正と機能向上 • EMQX をバージョン 4.4.9 に更新します。
1.0.1	一部の MQTT クライアントが接続に失敗する原因となる TLS ハンドシェイク中の問題を修正します。
1.0.0	当初のバージョン

nucleus テレメトリエミッタ

nucleus テレメトリエミッタコンポーネント (`aws.greengrass.telemetry.NucleusEmitter`) は、システムヘルステレメトリデータを収集して、ローカルトピックと AWS IoT Core MQTT トピックに継続的にパブリッシュします。このコンポーネントは、Greengrass コアデバイスでリアル タイム システム テレメトリの収集を可能にします。システムテレメトリデータを Amazon に発行する Greengrass テレメトリエージェントの詳細については EventBridge、「」を参照してくださいAWS IoT Greengrass コアデバイスからシステムヘルステレメトリデータを収集する。

デフォルトで、nucleus テレメトリエミッタコンポーネントは 60 秒ごとにテレメトリデータを次のローカルパブリッシュ/サブスクライブトピックにパブリッシュします。

```
$local/greengrass/telemetry
```

nucleus テレメトリエミッタコンポーネントは、デフォルトで AWS IoT Core MQTT トピックにパブリッシュしません。このコンポーネントをデプロイするとき、 AWS IoT Core MQTT トピックにパブリッシュするように設定できます。MQTT トピックを使用してデータを AWS クラウド にパブリッシュすることには、AWS IoT Core 価格設定が適用されます。

AWS IoT Greengrass はいくつかのコミュニティコンポーネントを提供し、InfluxDB と Grafana を使用してコアデバイスでローカルにテレメトリデータを分析と可視化するうえで役立ちます。これらのコンポーネントは、nucleus エミッタコンポーネントのテレメトリデータを使用します。詳細については、InfluxDB パブリッシャーコンポーネントの「README」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 1.0.x

タイプ

このコンポーネントはプラグインコンポーネント (`aws.greengrass.plugin`) です。Greengrass nucleus は、nucleus と同じ Java バーチャルマシン (JVM) でこのコンポーネントを実行します。コアデバイスでこのコンポーネントのバージョンを変更するとき、nucleus が再起動します。

このコンポーネントは、Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

1.0.8

次の表に、このコンポーネントのバージョン 1.0.8 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.4.0 <2.13.0	ハード

1.0.7

次の表に、このコンポーネントのバージョン 1.0.7 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.4.0 <2.12.0	ハード

1.0.6

次の表に、このコンポーネントのバージョン 1.0.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.4.0 <2.11.0	ハード

1.0.5

次の表に、このコンポーネントのバージョン 1.0.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.4.0 <2.10.0	ハード

1.0.4

次の表に、このコンポーネントのバージョン 1.0.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.4.0 <2.9.0	ハード

1.0.3

次の表に、このコンポーネントのバージョン 1.0.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.4.0 <2.8.0	ハード

1.0.2

次の表に、このコンポーネントのバージョン 1.0.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.4.0 <2.7.0	ハード

1.0.1

次の表に、このコンポーネントのバージョン 1.0.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.4.0 <2.6.0	ハード

1.0.0

次の表に、このコンポーネントのバージョン 1.0.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.4.0 <2.5.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

`pubSubPublish`

(オプション) テレメトリデータを `$local/greengrass/telemetry` トピックにパブリッシュするかどうか定義します。サポートされている値は、`true` および `false` です。

デフォルト: `true`

`mqttTopic`

(オプション) このコンポーネントがテレメトリデータをパブリッシュする AWS IoT Core MQTT トピック。

この値をテレメトリーデータをパブリッシュする AWS IoT Core MQTT トピックに設定します。この値が空のとき、nucleus エミッタはテレメトリデータを AWS クラウド にパブリッシュしません。

> **Note**
>
> MQTT トピックを使用してデータを AWS クラウド にパブリッシュすることには、AWS IoT Core 価格設定が適用されます。

デフォルト: `""`

`telemetryPublishIntervalMs`

(オプション) コンポーネントがテレメトリーデータをパブリッシュする間隔 (ミリ秒単位)。この値をサポートされている最小値よりも低く設定する場合、コンポーネントは代わりに最小値を使用します。

> **Note**
>
> パブリッシュ間隔を短くすると、コアデバイスの CPU 使用率が高くなります。デフォルトのパブリッシュ間隔から開始して、デバイスの CPU 使用率に基づいて調整することをお勧めします。

最小: `500`

デフォルト: `60000`

Example 例: 設定マージの更新

次の例は、テレメトリデータを 5 秒ごとに `$local/greengrass/telemetry` トピックと `greengrass/myTelemetry` AWS IoT Core MQTT トピックにパブリッシュできるようにする設定マージ更新のサンプルを示しています。

```
{
  "pubSubPublish": "true",
  "mqttTopic": "greengrass/myTelemetry",
  "telemetryPublishIntervalMs": 5000
}
```

出力データ

このコンポーネントは、次のトピックにテレメトリメトリクスをJSON 配列としてパブリッシュします。

ローカルトピック: `$local/greengrass/telemetry`

必要に応じて、テレメトリメトリクスを AWS IoT Core MQTT トピックにパブリッシュすることもできます。トピックの詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT トピック」を参照してください。

Example データの例

```
[
  {
    "A": "Average",
    "N": "CpuUsage",
    "NS": "SystemMetrics",
    "TS": 1627597331445,
    "U": "Percent",
    "V": 26.21981271562346
  },
  {
    "A": "Count",
    "N": "TotalNumberOfFDs",
    "NS": "SystemMetrics",
    "TS": 1627597331445,
    "U": "Count",
    "V": 7316
  },
  {
```

```
    "A": "Count",
    "N": "SystemMemUsage",
    "NS": "SystemMetrics",
    "TS": 1627597331445,
    "U": "Megabytes",
    "V": 10098
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsStarting",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsInstalled",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsStateless",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsStopping",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsBroken",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
```

```
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsRunning",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 7
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsErrored",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsNew",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsFinished",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 2
  }
]
```

出力配列には、次のプロパティを持つメトリクスのリストが含まれています。

A

メトリクスの集計タイプ。

CpuUsage メトリクスの場合、メトリクスのパブリッシュ値は、前回のパブリッシュイベント以降の平均 CPU 使用率であるため、このプロパティは Average に設定されます。

他のすべてのメトリクスの場合、nucleus エミッタはメトリクス値を集計せず、さらにこのプロパティは Count に設定されます。

N

メトリクスの名前。

NS

メトリクスの名前空間。

TS

データが収集された時刻のタイムスタンプ。

U

メトリクス値の単位。

V

メトリクスの値。

nucleus エミッタは、次のメトリクスを発行します。

名前	説明
システム	
SystemMemUsage	オペレーティングシステムを含む、Greengrass コアデバイスのすべてのアプリケーションで現在使用されているメモリの量。
CpuUsage	オペレーティングシステムを含む Greengrass コアデバイスのすべてのアプリケーションで現在使用されている CPU の量。

名前	説明
TotalNumberOfFDs	Greengrass コアデバイスのオペレーティングシステムによって保存されているファイルディスクリプタの数。1 つのファイルディスクリプタは、1 つのオープンファイルを一意に識別します。
Greengrass nucleus	
NumberOfComponentsRunning	Greengrass コアデバイスで実行されているコンポーネント数。
NumberOfComponentsErrored	Greengrass コアデバイスでエラー状態にあるコンポーネントの数。
NumberOfComponentsInstalled	Greengrass コアデバイスでインストールされているコンポーネントの数。
NumberOfComponentsStarting	Greengrass コアデバイスで開始されているコンポーネントの数。
NumberOfComponentsNew	Greengrass コアデバイスで新しくなったコンポーネントの数。
NumberOfComponentsStopping	Greengrass コアデバイスで停止しているコンポーネントの数。
NumberOfComponentsFinished	Greengrass コアデバイスで終了するコンポーネントの数。

名前	説明
NumberOfComponents Broken	Greengrass コアデバイスで壊れているコンポーネントの数。
NumberOfComponents Stateless	Greengrass コアデバイスでステートレスであるコンポーネントの数。

使用方法

システムヘルステレメトリデータを使用するには、nucleus エミッタがテレメトリデータをパブリッシュするトピックへサブスクライブし、必要に応じてそのデータに反応するカスタムコンポーネントを作成できます。nucleus エミッタコンポーネントには、テレメトリデータをローカルトピックにパブリッシュするオプションが用意されているため、そのトピックをサブスクライブし、パブリッシュされたデータを使用してコアデバイスでローカルに動作させることができます。次に、クラウドへの接続が制限されている場合でも、コアデバイスはテレメトリデータに反応できます。

例えば、テレメトリデータの `$local/greengrass/telemetry` トピックを参照し、ストリームマネージャーコンポーネントにデータを送信するようにコンポーネントを設定して、データを AWS クラウド にストリーミングできます。このようなコンポーネントの作成方法の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」と「ストリームマネージャーを使用するカスタムコンポーネントを作成する」を参照してください。

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
1.0.8	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
1.0.7	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
1.0.6	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
1.0.5	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
1.0.4	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
1.0.3	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。

バージョン	変更
1.0.2	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
1.0.1	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
1.0.0	当初のバージョン

PKCS#11 プロバイダ

PKCS#11 プロバイダコンポーネント (`aws.greengrass.crypto.Pkcs11Provider`) は、PKCS#11 インターフェイスを通してハードウェアセキュリティモジュール (HSM) を使用するため、AWS IoT Greengrass Core ソフトウェアを設定できるようにします。このコンポーネントは、証明書とプライベートキーファイルを安全に保存できるようにして、ソフトウェアで公開または複製されないようにします。詳細については、「ハードウェアセキュリティ統合」を参照してください。

HSM に証明書とプライベートキーを保存する Greengrass コアデバイスをプロビジョニングするには、AWS IoT Greengrass Core ソフトウェアをインストールするときに、このコンポーネントをプロビジョニングプラグインとして指定する必要があります。詳細については、「手動リソースプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。

AWS IoT Greengrass は、このコンポーネントをインストール中にプロビジョニングプラグインとして指定するためにダウンロードできる JAR ファイルとして提供します。次の URL としてコンポーネントの JAR ファイルの最新バージョンをダウンロードできます: https://d2s8p88vqu9w66.cloudfront.net/releases/Pkcs11Provider/aws.greengrass.crypto.Pkcs11Provider-latest.jar。

トピック

- バージョン
- タイプ
- オペレーティングシステム
- 要件
- 依存関係

- 構成
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.0.x

タイプ

このコンポーネントはプラグインコンポーネント (`aws.greengrass.plugin`) です。Greengrass nucleus は、nucleus と同じ Java バーチャルマシン (JVM) でこのコンポーネントを実行します。コアデバイスでこのコンポーネントのバージョンを変更するとき、nucleus が再起動します。

このコンポーネントは、Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、Linux コアデバイスにのみインストールできます。

要件

このコンポーネントには次の要件があります。

- PKCS#1 v1.5 署名スキームと RSA-2048 キーサイズ (またはそれ以上の規模) または ECC キーを備えた RSA キーをサポートするハードウェアセキュリティモジュール。

> **Note**
>
> ECC キーを備えたハードウェアセキュリティモジュールを使用するには、v2.5.6 以降の Greengrass nucleus を使用する必要があります。
> ハードウェアセキュリティモジュールとシークレットマネージャーを使用するには、RSA キーを備えたハードウェアセキュリティモジュールを使用する必要があります。

- PKCS#11 関数を呼び出すため、AWS IoT Greengrass Core ソフトウェアがランタイム時 (libdl を使用) にロードできる PKCS#11 プロバイダライブラリ。PKCS#11 プロバイダライブラリは、次の PKCS#11 API オペレーションを実装する必要があります。
  - C_Initialize
  - C_Finalize
  - C_GetSlotList
  - C_GetSlotInfo
  - C_GetTokenInfo
  - C_OpenSession
  - C_GetSessionInfo
  - C_CloseSession
  - C_Login
  - C_Logout
  - C_GetAttributeValue
  - C_FindObjectsInit
  - C_FindObjects
  - C_FindObjectsFinal
  - C_DecryptInit
  - C_Decrypt
  - C_DecryptUpdate
  - C_DecryptFinal
  - C_SignInit
  - C_Sign
  - C_SignUpdate
  - C_SignFinal
  - C_GetMechanismList
  - C_GetMechanismInfo
  - C_GetInfo
  - C_GetFunctionList
- ハードウェアモジュールは、「PKCS#11 仕様」で定義されているスロットラベルで解決できる必要があります。

- プライベートキーと証明書は HSM の同じスロットに保存する必要があり、HSM がオブジェクト ID をサポートしている場合、同じオブジェクトラベルとオブジェクト ID を使用する必要があります。
- 証明書とプライベートキーがオブジェクトラベルで解決できる必要があります。
- プライベートキーには、次の許可が必要です。
  - `sign`
  - `decrypt`
- (オプション) シークレットマネージャーコンポーネントを使用する場合、バージョン 2.1.0 以降を使用する必要があります。また、プライベートキーには次の許可が必要です。
  - `unwrap`
  - `wrap`
- (オプション) TPM2 ライブラリを使用し、Greengrass コアをサービスとして実行している場合は、環境変数に PKCS#11 ストアの場所を指定する必要があります。次の例は、必要な環境変数を含む systemd サービスファイルです。

```
[Unit]
Description=Greengrass Core
After=network.target

[Service]
Type=simple
PIDFile=/var/run/greengrass.pid
Environment=TPM2_PKCS11_STORE=/path/to/store/directory
RemainAfterExit=no
Restart=on-failure
RestartSec=10
ExecStart=/bin/sh /greengrass/v2/alts/current/distro/bin/loader

[Install]
WantedBy=multi-user.target
```

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定

義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.0.7

次の表に、このコンポーネントのバージョン 2.0.7 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.3 <2.13.0	ソフト

2.0.6

次の表に、このコンポーネントのバージョン 2.0.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.3 <2.12.0	ソフト

2.0.5

次の表に、このコンポーネントのバージョン 2.0.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.3 <2.11.0	ソフト

2.0.4

次の表に、このコンポーネントのバージョン 2.0.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.3 <2.10.0	ソフト

2.0.3

次の表に、このコンポーネントのバージョン 2.0.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.3 <2.9.0	ソフト

2.0.2

次の表に、このコンポーネントのバージョン 2.0.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.3 <2.8.0	ソフト

2.0.1

次の表に、このコンポーネントのバージョン 2.0.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.3 <2.7.0	ソフト

2.0.0

次の表に、このコンポーネントのバージョン 2.0.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.3 <2.6.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

`name`

PKCS#11 設定の名前。

`library`

AWS IoT Greengrass Core ソフトウェアが libdl でロードできる PKCS#11 実装のライブラリへの絶対ファイルパス

`slot`

プライベートキーとデバイス証明書を含むスロットの ID。この値は、スロットインデックスやスロットラベルとは異なります。

`userPin`

スロットへのアクセスに使用するユーザー PIN。

Example 例: 設定マージの更新

```
{
  "name": "softhsm_pkcs11",
  "library": "/usr/lib/softhsm/libsofthsm2.so",
  "slot": 1,
  "userPin": "1234"
}
```

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.0.7	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.0.6	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.0.5	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.0.4	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.0.3	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.0.2	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.0.1	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.0.0	当初のバージョン

シークレットマネージャー

シークレットマネージャーコンポーネント (`aws.greengrass.SecretManager`) は、AWS Secrets Manager から Greengrass コアデバイスにシークレットをデプロイします。このコンポーネントを使用して、Greengrass コアデバイスのカスタムコンポーネントでパスワードなどの認証情報を安全に使用します。Secrets Manager の詳細については、「AWS Secrets Manager ユーザーガイド」の「AWS Secrets Manager とは」を参照してください。

カスタム Greengrass コンポーネントのこのコンポーネントのシークレットにアクセスするには、の GetSecretValueオペレーションを使用しますAWS IoT Device SDK。詳細については、「AWS IoT Device SDK を使用して Greengrass nucleus、その他のコンポーネント、および AWS IoT Core と通信する と シークレット値を取得する」を参照してください。

このコンポーネントは、コアデバイスのシークレットを暗号化して、認証情報およびパスワードを使用する必要があるまで安全に保ちます。コアデバイスのプライベートキーを使用して、シークレットを暗号化および復号化します。

トピック

- バージョン
- タイプ
- オペレーティングシステム
- 要件
- 依存関係
- 構成
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x
- 2.0.x

タイプ

このコンポーネントはプラグインコンポーネント (`aws.greengrass.plugin`) です。Greengrass nucleus は、nucleus と同じ Java バーチャルマシン (JVM) でこのコンポーネントを実行します。コアデバイスでこのコンポーネントのバージョンを変更するとき、nucleus が再起動します。

このコンポーネントは、Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- 次の IAM ポリシーの例で示されているように、Greengrass デバイスのロールは `secretsmanager:GetSecretValue` アクションを許可する必要があります。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "secretsmanager:GetSecretValue"
```

```
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:secretsmanager:region:123456789012:secret:MySecret"
      ]
    }
  ]
}
```

> **Note**
>
> カスタマー管理の AWS Key Management Service キーを使用してシークレットを暗号化する場合は、デバイスのロールで `kms:Decrypt` アクションも許可する必要があります。

Secrets Manager の IAMポリシーの詳細については、「AWS Secrets Manager ユーザーガイド」の以下を参照してください。

- [Authentication and access control for AWS Secrets Manager] (の認証とアクセスコントロール)
- [Actions, resources, and context keys you can use in an IAM policy or secret policy for AWS Secrets Manager] (の IAM ポリシーまたはシークレットポリシーで使用できるアクション、リソース、およびコンテキストキー)

- カスタムコンポーネントは、このコンポーネントで保存したシークレットを `aws.greengrass#GetSecretValue` が取得できるようにするための認可ポリシーを定義する必要があります。この認可ポリシーでは、コンポーネントのアクセスを特定のシークレットに制限できます。詳細については、「シークレットマネージャー IPC 認証」を参照してください。
- (オプション) コアデバイスのプライベートキーと証明書を [hardware security module] (ハードウェアセキュリティモジュール) (HSM) に保存する場合、HSM は RSA キーをサポートし、プライベートキーには `unwrap` アクセス許可が必要で、パブリックキーには `wrap` アクセス許可が必要です。

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
secretsmanager. *region*.amazonaws.com	443	はい	コアデバイスにシークレットをダウンロードします。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.13.0	ソフト

2.1.6

次の表に、このコンポーネントのバージョン 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.12.0	ソフト

2.1.5

次の表に、このコンポーネントのバージョン 2.1.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.11.0	ソフト

2.1.4

次の表に、このコンポーネントのバージョン 2.1.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.10.0	ソフト

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.9.0	ソフト

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.8.0	ソフト

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.7.0	ソフト

2.1.0

次の表に、このコンポーネントのバージョン 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.6.0	ソフト

2.0.9

次の表に、このコンポーネントのバージョン 2.0.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト

2.0.8

次の表に、このコンポーネントのバージョン 2.0.8 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト

2.0.7

次の表に、このコンポーネントのバージョン 2.0.7 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト

2.0.6

次の表に、このコンポーネントのバージョン 2.0.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト

2.0.4 and 2.0.5

次の表に、このコンポーネントのバージョン 2.0.4 および 2.0.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.3 <2.1.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

`cloudSecrets`

コアデバイスにデプロイする Secrets Manager シークレットのリスト。ラベルを指定して、デプロイする各シークレットのバージョンを定義できます。バージョンを指定しない場合、このコンポーネントはステージングラベル `AWSCURRENT` がアタッチされたバージョンをデプロイします。詳細については、「AWS Secrets Manager ユーザーガイド」の「ステージングラベル」を参照してください。

シークレットマネージャーコンポーネントは、シークレットをローカルにキャッシュします。Secrets Manager でシークレット値が変更された場合、このコンポーネントは新しい値を自動的に取得しません。ローカルコピーを更新するには、シークレットに新しいラベルを付け、新しいラベルで識別されるシークレットを取得するようにこのコンポーネントを設定します。

各オブジェクトには、次の情報が含まれます:

`arn`

デプロイするシークレットの ARN。シークレットの ARN は、完全な ARN でも部分的な ARN でもかまいません。部分的な ARN ではなく、完全な ARN を指定することをお勧めします。詳細については、「部分的な ARN からのシークレットの検索」を参照してください。以下は、完全な ARN と部分的な ARN の例です。

- 完全な ARN: `arn:aws:secretsmanager:us-east-2:111122223333:secret:SecretName-abcdef`
- 部分的な ARN: `arn:aws:secretsmanager:us-east-2:111122223333:secret:SecretName`

`labels`

(オプション) コアデバイスにデプロイするシークレットのバージョンを識別するためのラベルのリスト。

各ラベルは文字列である必要があります。

Example 例: 設定マージの更新

```
{
  "cloudSecrets": [
    {
      "arn": "arn:aws:secretsmanager:us-west-2:123456789012:secret:MyGreengrassSecret-abcdef"
    }
  ]
}
```

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.7	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.1.6	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.5	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.4	バグ修正と機能向上 シークレットマネージャーがデプロイされ、Greengrass nucleus が再起動されたときに、キャッシュされたシークレットが削除される問題を修正しました。 Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.1.3	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.1.1	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.0	新機能 • ハードウェアセキュリティ統合のサポートが追加されました。シークレットマネージャーコンポーネントは、ハードウェアセキュリティモジュール (HSM) に保存するプライベートキーを使用して、シークレットを暗号化および復号化できます。詳細については、「ハードウェアセキュリティ統合」を参照してください。 バグ修正と機能向上 • Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.0.9	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.0.8	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.0.7	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.0.6	Greengrass nucleus バージョン 2.1.0 のリリース用にバージョンが更新されました。
2.0.5	改良点 • AWS 中国リージョンと AWS GovCloud (US) リージョンのサポートを追加します。
2.0.4	当初のバージョン

セキュアトンネリング

`aws.greengrass.SecureTunneling` コンポーネントにより、制限されたファイアウォールの背後にある Greengrass コアデバイスとの、セキュアな双方向通信を確立できます。

例えば、ファイアウォールの背後にあり、すべての着信接続を禁止している、Greengrass を使用しているとします。セキュアトンネリングは、MQTT を使用してデバイスにアクセストークンを転送し、 を使用してファイアウォールを介してデバイスへの SSH 接続 WebSockets を行います。この AWS IoT マネージドのトンネルを使用することで、必要な SSH 接続をデバイスに対し開くことができます。AWS IoT セキュアトンネリングを使用してリモートデバイスに接続する方法の詳細については、「AWS IoT デベロッパーガイド」の「AWS IoT セキュアトンネリング」を参照してください。

このコンポーネントは、`$aws/things/greengrass-core-device/tunnels/notify`トピックで AWS IoT Core MQTT メッセージブローカーをサブスクライブし、安全なトンネリング通知を受信します。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 1.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、Linux コアデバイスにのみインストールできます。

アーキテクチャ:

- Armv71
- Armv8 (AArch64)
- x86_64

要件

このコンポーネントには次の要件があります。

- セキュアトンネリングコンポーネントで使用できる最低 32 MB のディスク容量。この要件には、同じデバイスで実行されている Greengrass コアソフトウェアやその他のコンポーネントは含まれません。
- セキュアトンネリングコンポーネントで使用できる最小 16 MB RAM。この要件には、同じデバイスで実行されている Greengrass コアソフトウェアやその他のコンポーネントは含まれません。詳細については、「JVM オプションでメモリ割り当てを制御する」を参照してください。
- セキュアトンネリングコンポーネントバージョン 1.0.12 以降では、Linux カーネルが 3.2 以降の GNU C ライブラリ (glibc) バージョン 2.25 以降が必要です。長期サポート終了日を過ぎたオペレーティングシステムとライブラリのバージョンはサポートされていません。長期サポートを備えたオペレーティングシステムとライブラリを使用する必要があります。
- オペレーティングシステムと Java ランタイムの両方を 64 ビットとしてインストールする必要があります。
- Greengrass コアデバイスにインストールされ、PATH 環境変数に追加された Python 3.5 以降。
- `libcrypto.so.1.1` は、Greengrass コアデバイスにインストールされ、PATH 環境変数に追加されています。
- Greengrass コアデバイスのポート 443 で、アウトバウンドトラフィックを開きます。

- Greengrass コアデバイスとの通信に必要な、通信サービスのサポートをオンにします。例えば、デバイスへの SSH 接続を開くには、そのデバイスで SSH を有効にする必要があります。

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
`data.tunneling.iot.` *`region`*`.amazonaws.com`	443	はい	セキュアトンネルを確立します。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

1.0.18

次の表に、このコンポーネントのバージョン 1.0.18 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト

1.0.16 – 1.0.17

次の表に、このコンポーネントのバージョン 1.0.16 から 1.0.17 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト

1.0.14 – 1.0.15

次の表に、このコンポーネントのバージョン 1.0.14 から 1.0.15 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト

1.0.11 – 1.0.13

次の表に、このコンポーネントのバージョン 1.0.11 ~ 1.0.13 の間の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト

1.0.10

次の表に、このコンポーネントのバージョン 1.0.10 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト

1.0.9

次の表に、このコンポーネントのバージョン 1.0.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト

1.0.8

次の表に、このコンポーネントのバージョン 1.0.8 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト

1.0.5 - 1.0.7

次の表に、このコンポーネントのバージョン 1.0.5 から 1.0.7 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト

1.0.4

次の表に、このコンポーネントのバージョン 1.0.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト

1.0.3

次の表に、このコンポーネントのバージョン 1.0.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト

1.0.2

次の表に、このコンポーネントのバージョン 1.0.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト

1.0.1

次の表に、このコンポーネントのバージョン 1.0.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト

1.0.0

次の表に、このコンポーネントのバージョン 1.0.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.3 <2.1.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

`OS_DIST_INFO`

(オプション) コアデバイスのオペレーティングシステム。デフォルトでコンポーネントは、コアデバイスで実行されているオペレーティングシステムを自動的に識別しようとします。コンポーネントがデフォルト値で開始できない場合は、この値を使用してオペレーティングシステムを指定します。このコンポーネントでサポートされているオペレーティングシステムのリストについては、「デバイスの要件」を参照してください。

これには、次のいずれかの値を指定できます: `auto`、`ubuntu`、`amzn2`、`raspberrypi`。

デフォルト: `auto`

`accessControl`

(オプション) コンポーネントがセキュアトンネリング通知トピックにサブスクライブできるようにする [authorization policy] (認可ポリシー) を含むオブジェクト。

> **Note**
>
> デプロイがモノグループをターゲットとする場合は、この設定パラメータを変更しないでください。デプロイで、個別のコアデバイスを対象としており、そのサブスクリプションをデバイスのトピックに制限したい場合は、そのコアデバイスのモノの名前 (thing name) を指定します。デバイスの認証ポリシー内にある `resources` の値で、MQTT のトピックワイルドカードを、対象のデバイスのモノの名前に置き換えます。

```
{
  "aws.greengrass.ipc.mqttproxy": {
    "aws.iot.SecureTunneling:mqttproxy:1": {
      "policyDescription": "Access to tunnel notification pubsub topic",
      "operations": [
        "aws.greengrass#SubscribeToIoTCore"
      ],
      "resources": [
        "$aws/things/+/tunnels/notify"
      ]
    }
  }
}
```

Example 例: 設定マージの更新

次の設定例では、このコンポーネントが Ubuntu を実行する **MyGreengrassCore** という名前のコアデバイスでセキュアなトンネルを開くことができるように指定しています。

```
{
  "OS_DIST_INFO": "ubuntu",
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
      "aws.iot.SecureTunneling:mqttproxy:1": {
        "policyDescription": "Access to tunnel notification pubsub topic",
```

```
        "operations": [
          "aws.greengrass#SubscribeToIoTCore"
        ],
        "resources": [
          "$aws/things/MyGreengrassCore/tunnels/notify"
        ]
      }
    }
  }
}
```

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

```
/greengrass/v2/logs/aws.greengrass.SecureTunneling.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。をAWS IoT Greengrassルートフォルダへのパス*/greengrass/v2*に置き換えます。

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.SecureTunneling.log
  ```

ライセンス

このコンポーネントには、次のサードパーティーソフトウェア/ライセンス品が含まれています。

- AWS IoT デバイスクライアント/Apache License 2.0
- AWS IoT Device SDK for Java/Apache License 2.0
- gson/Apache License 2.0
- log4j/Apache License 2.0
- slf4j/Apache License 2.0

使用方法

デバイスでセキュアトンネリングコンポーネントを使用するには、次の手順を実行します。

1. セキュアトンネリングコンポーネントをデバイスにデプロイします。
2. AWS IoT コンソールを開きます。左側のメニューからリモートアクション を選択し、セキュアトンネル を選択します。
3. Greengrass デバイスへのトンネルを作成します。
4. ソースアクセストークンをダウンロードします。
5. 送信元アクセストークンでローカルプロキシを使用して送信先に接続します。詳細については、「 AWS IoTデベロッパーガイド」の「ローカルプロキシの使用方法」を参照してください。

以下も参照してください。

- AWS IoT 「 デベロッパーガイド」の「 セキュアトンネリングAWS IoT
- 「 デベロッパーガイド」の「ローカルプロキシの使用方法AWS IoT」

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
1.0.18	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
1.0.17	バグ修正と機能向上 • ユーザーによるトンネルの作成を妨げていたスレッドクリーンアップの問題を修正しました。これで、このコンポーネントは CloseTunnel シグナルを受信した後、またはトンネルが 12 時間後に期限切れになった場合に、スレッドをクリーンアップします。
1.0.16	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
1.0.15	バグ修正と機能向上 • デバイスにホームディレクトリがないユーザーのスタートアップの問題を修正しました。セキュアトンネリングコンポーネントは、シャドウドキュメント用のディレクトリを作成せずに起動するようになりました。

バージョン	変更
1.0.14	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
1.0.13	バグ修正と機能向上 • 複数のトンネルでデバイスをターゲットにすることを、孤立したクライアントプロセスが妨げていた問題を修正しました。
1.0.12	バグ修正と機能向上 • Raspberry Pi OS 上で実行する場合の x86_64 (AMD64) と ARMv8 (Aarch64) のサポートが追加されました。
1.0.11	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
1.0.10	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
1.0.9	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
1.0.8	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
1.0.7	バグ修正と機能向上 • SCP 経由で大きなファイルを転送すると、コンポーネントが切断される問題を修正しました。
1.0.6	このバージョンには、バグ修正が含まれています。
1.0.5	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
1.0.4	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
1.0.3	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。

バージョン	変更
1.0.2	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
1.0.1	Greengrass nucleus バージョン 2.1.0 のリリース用にバージョンが更新されました。
1.0.0	当初のバージョン

シャドウマネージャー

シャドウマネージャーコンポーネント (`aws.greengrass.ShadowManager`) を使用すると、コアデバイスでローカルシャドウサービスを有効にできます。ローカルシャドウサービスを使用すると、コンポーネントはプロセス間通信を使用してローカルシャドウとやり取りすることができます。シャドウマネージャーコンポーネントは、ローカルシャドウドキュメントの保存を管理し、ローカルシャドウ状態の AWS IoT Device Shadow サービスとの同期も処理します。

Greengrass コアデバイスがシャドウとやり取りする方法の詳細については、「デバイスシャドウとやり取り」を参照してください。

トピック

- バージョン
- タイプ
- オペレーティングシステム
- 要件
- 依存関係
- 構成
- ローカルログファイル
- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.3.x

- 2.2.x
- 2.1.x
- 2.0.x

タイプ

このコンポーネントはプラグインコンポーネント (`aws.greengrass.plugin`) です。Greengrass nucleus は、nucleus と同じ Java バーチャルマシン (JVM) でこのコンポーネントを実行します。コアデバイスでこのコンポーネントのバージョンを変更するとき、nucleus が再起動します。

このコンポーネントは、Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- （オプション) シャドウを AWS IoT Device Shadow サービスに同期するには、Greengrass コアデバイスの AWS IoT ポリシーで次の AWS IoT Core シャドウポリシーアクションを許可する必要があります。
  - `iot:GetThingShadow`
  - `iot:UpdateThingShadow`
  - `iot:DeleteThingShadow`

  これらの AWS IoT Core ポリシーの詳細については、「 AWS IoT デベロッパーガイド」の「AWS IoT Core ポリシーアクション」を参照してください。

最小 AWS IoT ポリシーの詳細については、「」を参照してください。 AWS IoT Greengrass V2 コアデバイス向けの最低限の AWS IoT ポリシー

- シャドウマネージャーコンポーネントは、VPC での実行がサポートされています。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.3.5 – 2.3.6

次の表に、このコンポーネントのバージョン 2.3.5 および 2.3.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.13.0	ソフト

2.3.3 and 2.3.4

次の表に、このコンポーネントのバージョン 2.3.3 および 2.3.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.12.0	ソフト

2.3.2

次の表に、このコンポーネントのバージョン 2.3.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.11.0	ソフト

2.3.0 and 2.3.1

次の表に、このコンポーネントのバージョン 2.3.0 および 2.3.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.5.0 <2.10.0	ソフト

2.2.3 and 2.2.4

次の表に、このコンポーネントのバージョン 2.2.3 および 2.2.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <3.0.0	ソフト

2.2.2

次の表に、このコンポーネントのバージョン 2.2.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.9.0	ソフト

2.2.1

次の表に、このコンポーネントのバージョン 2.2.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.8.0	ソフト

2.1.1 and 2.2.0

次の表に、このコンポーネントのバージョン 2.1.1 および 2.2.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.7.0	ソフト

2.0.5 - 2.1.0

次の表に、このコンポーネントのバージョン 2.0.5 から 2.1.0 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.6.0	ソフト

2.0.3 and 2.0.4

次の表に、このコンポーネントのバージョン 2.0.3 および 2.0.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.5.0	ソフト

2.0.1 and 2.0.2

次の表に、このコンポーネントのバージョン 2.0.1 および 2.0.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.4.0	ソフト

2.0.0

次の表に、このコンポーネントのバージョン 2.0.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.2.0 <2.3.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

2.3.x

`strategy`

(オプション) このコンポーネントが AWS IoT Core とコアデバイス間でシャドウを同期するために使用する戦略。

このオブジェクトには、次の情報が含まれます。

`type`

(オプション) このコンポーネントが AWS IoT Core とコアデバイス間でシャドウを同期させるために使用する戦略のタイプ。次のオプションから選択します。

- `realTime` - シャドウの更新が発生する AWS IoT Core たびにシャドウを と同期します。
- `periodic` - `delay`設定パラメータ AWS IoT Core で指定した一定の間隔でシャドウをと同期します。

デフォルト: `realTime`

`delay`

(オプション) `periodic` 同期戦略を指定した場合に、このコンポーネントがシャドウを AWS IoT Core と同期する間隔 (秒単位)。

> **Note**
>
> `periodic` 同期戦略を指定する場合、このパラメータは必須です。

`synchronize`

(オプション) シャドウを AWS クラウド と同期する方法を決定する同期設定。

> **Note**
> シャドウを AWS クラウド と同期させるには、このプロパティを使用して設定アップデートを作成する必要があります。

このオブジェクトには、次の情報が含まれます。

`coreThing`

(オプション) 同期するコアデバイスシャドウ。このオブジェクトには、次の情報が含まれます。

`classic`

(オプション) デフォルトでは、シャドウマネージャーはコアデバイスのクラシックシャドウのローカル状態を AWS クラウド と同期させます。クラシックデバイスシャドウを同期しない場合は、これを `false` に設定します。

デフォルト: `true`

`namedShadows`

(オプション) 同期する名前付きコアデバイスシャドウのリスト。シャドウの正確な名前を指定する必要があります。

> **Warning**
> AWS IoT Greengrass サービス
> は、`AWSManagedGreengrassV2Deployment`名前付きシャドウを使用して、個々のコアデバイスを対象とするデプロイを管理します。この名前付きシャドウは、 AWS IoT Greengrass サービスで使用するために予約されています。この名前付きシャドウを更新または削除しないでください。

`shadowDocumentsMap`

(オプション) 同期する追加のデバイスシャドウ。この構成パラメータを使用すると、シャドウドキュメントの指定が簡単になります。`shadowDocuments` オブジェクトの代わりに、このパラメータを使用することをお勧めします。

> **Note**
>
> `shadowDocumentsMap` オブジェクトを指定する場合は、`shadowDocuments` オブジェクトを指定しないでください。

各オブジェクトには、次の情報が含まれます:

`thingName`

このシャドウ構成の *`[thingName]`* のシャドウ設定。

`classic`

(オプション) `thingName` デバイスのクラシックデバイスシャドウを同期しない場合は、これを `false` に設定します。

`namedShadows`

同期する名前付きシャドウのリスト。シャドウの正確な名前を指定する必要があります。

`shadowDocuments`

(オプション) 同期する追加のデバイスシャドウのリスト。代わりに `shadowDocumentsMap` パラメータを使用することをお勧めします。

> **Note**
>
> `shadowDocuments` オブジェクトを指定する場合は、`shadowDocumentsMap` オブジェクトを指定しないでください。

このリストの各オブジェクトには、次の情報が含まれます。

`thingName`

シャドウを同期するデバイスのモノの名前。

`classic`

(オプション) `thingName` デバイスのクラシックデバイスシャドウを同期しない場合は、これを `false` に設定します。

デフォルト: `true`

`namedShadows`

(オプション) 同期する名前付きデバイスシャドウのリスト。シャドウの正確な名前を指定する必要があります。

`direction`

(オプション) ローカルシャドウサービスと AWS クラウド の間でシャドウを同期させる方向。このオプションを設定すると、 AWS クラウド への帯域幅と接続数を低減できます。次のオプションから選択します。

- `betweenDeviceAndCloud` – ローカルシャドウサービスと AWS クラウド を同期させる。
- `deviceToCloud` – シャドウ更新をローカルシャドウサービスから に送信し AWS クラウド、 からのシャドウ更新を無視します AWS クラウド。
- `cloudToDevice` – AWS クラウド からシャドウアップデートを受信し、ローカルシャドウサービスから AWS クラウド にシャドウアップデートを送信しない。

デフォルト: `BETWEEN_DEVICE_AND_CLOUD`

`rateLimits`

(オプション) シャドウサービス要求のレート制限を決定する設定。

このオブジェクトには、次の情報が含まれます。

`maxOutboundSyncUpdatesPerSecond`

(オプション) デバイスが送信する 1 秒あたりの同期要求の最大数。

デフォルト: 100 要求/秒

`maxTotalLocalRequestsRate`

(オプション) コアデバイスに送信される 1 秒あたりのローカル IPC 要求の最大数。

デフォルト: 200 要求/秒

`maxLocalRequestsPerSecondPerThing`

(オプション) 接続された IoT モノごとに送信される 1 秒あたりのローカル IPC 要求の最大数。

デフォルト: 1 件につき 20 要求/秒

> **Note**
>
> これらのレート制限パラメータは、ローカルシャドウサービスの 1 秒あたりの最大要求数を定義します。 AWS IoT Device Shadow サービスの 1 秒あたりの最大リクエスト数は、 によって異なります AWS リージョン。詳しくは、「Amazon Web Services 全般のリファレンス」の「AWS IoT デバイスシャドウサービス API」の制限を参照ください。

`shadowDocumentSizeLimitBytes`

(オプション) ローカルシャドウの各 JSON 状態ドキュメントの最大許容サイズ。

この値を大きくすると、クラウドシャドウの JSON 状態ドキュメントのリソース制限も増やす必要があります。詳しくは、「Amazon Web Services 全般のリファレンス」の「AWS IoT デバイスシャドウサービス API」の制限を参照ください。

デフォルト: 8,192 バイト

最大: 30720 バイト

Example 例: 設定マージの更新

次の例は、シャドウマネージャーコンポーネントで利用可能なすべての設定パラメータを使用した設定マージ更新のサンプルを示しています。

```
{
   "strategy":{
      "type":"periodic",
      "delay":300
   },
   "synchronize":{
      "shadowDocumentsMap":{
         "MyDevice1":{
            "classic":false,
```

```
            "namedShadows":[
               "MyShadowA",
               "MyShadowB"
            ]
         },
         "MyDevice2":{
            "classic":true,
            "namedShadows":[]
         }
      },
      "direction":"betweenDeviceAndCloud"
   },
   "rateLimits":{
      "maxOutboundSyncUpdatesPerSecond":100,
      "maxTotalLocalRequestsRate":200,
      "maxLocalRequestsPerSecondPerThing":20
   },
   "shadowDocumentSizeLimitBytes":8192
}
```

2.2.x

`strategy`

(オプション) このコンポーネントが AWS IoT Core とコアデバイス間でシャドウを同期するために使用する戦略。

このオブジェクトには、次の情報が含まれます。

`type`

(オプション) このコンポーネントが AWS IoT Core とコアデバイス間でシャドウを同期させるために使用する戦略のタイプ。次のオプションから選択します。

- `realTime` - シャドウの更新が発生する AWS IoT Core たびにシャドウを と同期します。
- `periodic` - `delay`設定パラメータ AWS IoT Core で指定した一定の間隔でシャドウを と同期します。

デフォルト: `realTime`

`delay`

(オプション) `periodic` 同期戦略を指定した場合に、このコンポーネントがシャドウを AWS IoT Core と同期する間隔 (秒単位)。

> **Note**
>
> `periodic` 同期戦略を指定する場合、このパラメータは必須です。

`synchronize`

(オプション) シャドウを AWS クラウド と同期する方法を決定する同期設定。

> **Note**
>
> シャドウを AWS クラウド と同期させるには、このプロパティを使用して設定アップデートを作成する必要があります。

このオブジェクトには、次の情報が含まれます。

`coreThing`

(オプション) 同期するコアデバイスシャドウ。このオブジェクトには、次の情報が含まれます。

`classic`

(オプション) デフォルトでは、シャドウマネージャーはコアデバイスのクラシックシャドウのローカル状態を AWS クラウド と同期させます。クラシックデバイスシャドウを同期しない場合は、これを `false` に設定します。

デフォルト: `true`

`namedShadows`

(オプション) 同期する名前付きコアデバイスシャドウのリスト。シャドウの正確な名前を指定する必要があります。

> **Warning**
>
> AWS IoT Greengrass サービス
> は、`AWSManagedGreengrassV2Deployment`名前付きシャドウを使用して、個々のコアデバイスを対象とするデプロイを管理します。この名前付きシャドウは、 AWS IoT Greengrass サービスで使用するために予約されています。この名前付きシャドウを更新または削除しないでください。

`shadowDocumentsMap`

(オプション) 同期する追加のデバイスシャドウ。この構成パラメータを使用すると、シャドウドキュメントの指定が簡単になります。`shadowDocuments` オブジェクトの代わりに、このパラメータを使用することをお勧めします。

> **Note**
>
> `shadowDocumentsMap` オブジェクトを指定する場合は、`shadowDocuments` オブジェクトを指定しないでください。

各オブジェクトには、次の情報が含まれます:

`thingName`

このシャドウ構成の *`[thingName]`* のシャドウ設定。

`classic`

(オプション) `thingName` デバイスのクラシックデバイスシャドウを同期しない場合は、これを `false` に設定します。

`namedShadows`

同期する名前付きシャドウのリスト。シャドウの正確な名前を指定する必要があります。

`shadowDocuments`

(オプション) 同期する追加のデバイスシャドウのリスト。代わりに `shadowDocumentsMap` パラメータを使用することをお勧めします。

> **Note**
>
> `shadowDocuments` オブジェクトを指定する場合は、`shadowDocumentsMap` オブジェクトを指定しないでください。

このリストの各オブジェクトには、次の情報が含まれます。

`thingName`

シャドウを同期するデバイスのモノの名前。

`classic`

(オプション) `thingName` デバイスのクラシックデバイスシャドウを同期しない場合は、これを `false` に設定します。

デフォルト: `true`

`namedShadows`

(オプション) 同期する名前付きデバイスシャドウのリスト。シャドウの正確な名前を指定する必要があります。

`direction`

(オプション) ローカルシャドウサービスと AWS クラウド の間でシャドウを同期させる方向。このオプションを設定すると、 AWS クラウド への帯域幅と接続数を低減できます。次のオプションから選択します。

- `betweenDeviceAndCloud` – ローカルシャドウサービスと AWS クラウド を同期させる。
- `deviceToCloud` – シャドウ更新をローカルシャドウサービスから に送信し AWS クラウド、 からのシャドウ更新を無視します AWS クラウド。
- `cloudToDevice` – AWS クラウド からシャドウアップデートを受信し、ローカルシャドウサービスから AWS クラウド にシャドウアップデートを送信しない。

デフォルト: `BETWEEN_DEVICE_AND_CLOUD`

`rateLimits`

(オプション) シャドウサービス要求のレート制限を決定する設定。

このオブジェクトには、次の情報が含まれます。

`maxOutboundSyncUpdatesPerSecond`

(オプション) デバイスが送信する 1 秒あたりの同期要求の最大数。

デフォルト: 100 要求/秒

`maxTotalLocalRequestsRate`

(オプション) コアデバイスに送信される 1 秒あたりのローカル IPC 要求の最大数。

デフォルト: 200 要求/秒

`maxLocalRequestsPerSecondPerThing`

(オプション) 接続された IoT モノごとに送信される 1 秒あたりのローカル IPC 要求の最大数。

デフォルト: 1 件につき 20 要求/秒

> **Note**
>
> これらのレート制限パラメータは、ローカルシャドウサービスの 1 秒あたりの最大要求数を定義します。 AWS IoT Device Shadow サービスの 1 秒あたりの最大リクエスト数は、 によって異なります AWS リージョン。詳しくは、「Amazon Web Services 全般のリファレンス」の「AWS IoT デバイスシャドウサービス API」の制限を参照ください。

`shadowDocumentSizeLimitBytes`

(オプション) ローカルシャドウの各 JSON 状態ドキュメントの最大許容サイズ。

この値を大きくすると、クラウドシャドウの JSON 状態ドキュメントのリソース制限も増やす必要があります。詳しくは、「Amazon Web Services 全般のリファレンス」の「AWS IoT デバイスシャドウサービス API」の制限を参照ください。

デフォルト: 8,192 バイト

最大: 30720 バイト

Example 例: 設定マージの更新

次の例は、シャドウマネージャーコンポーネントで利用可能なすべての設定パラメータを使用した設定マージ更新のサンプルを示しています。

```
{
   "strategy":{
      "type":"periodic",
      "delay":300
   },
   "synchronize":{
      "shadowDocumentsMap":{
         "MyDevice1":{
            "classic":false,
```

```
            "namedShadows":[
               "MyShadowA",
               "MyShadowB"
            ]
         },
         "MyDevice2":{
            "classic":true,
            "namedShadows":[]
         }
      },
      "direction":"betweenDeviceAndCloud"
   },
   "rateLimits":{
      "maxOutboundSyncUpdatesPerSecond":100,
      "maxTotalLocalRequestsRate":200,
      "maxLocalRequestsPerSecondPerThing":20
   },
   "shadowDocumentSizeLimitBytes":8192
}
```

2.1.x

`strategy`

(オプション) このコンポーネントが AWS IoT Core とコアデバイス間でシャドウを同期するために使用する戦略。

このオブジェクトには、次の情報が含まれます。

`type`

(オプション) このコンポーネントが AWS IoT Core とコアデバイス間でシャドウを同期させるために使用する戦略のタイプ。次のオプションから選択します。

- `realTime` - シャドウの更新が発生する AWS IoT Core たびにシャドウを と同期します。
- `periodic` - `delay`設定パラメータ AWS IoT Core で指定した一定の間隔でシャドウを と同期します。

デフォルト: `realTime`

`delay`

(オプション) `periodic` 同期戦略を指定した場合に、このコンポーネントがシャドウを AWS IoT Core と同期する間隔 (秒単位)。

> **Note**
>
> `periodic` 同期戦略を指定する場合、このパラメータは必須です。

`synchronize`

(オプション) シャドウを AWS クラウド と同期する方法を決定する同期設定。

> **Note**
>
> シャドウを AWS クラウド と同期させるには、このプロパティを使用して設定アップデートを作成する必要があります。

このオブジェクトには、次の情報が含まれます。

`coreThing`

(オプション) 同期するコアデバイスシャドウ。このオブジェクトには、次の情報が含まれます。

`classic`

(オプション) デフォルトでは、シャドウマネージャーはコアデバイスのクラシックシャドウのローカル状態を AWS クラウド と同期させます。クラシックデバイスシャドウを同期しない場合は、これを `false` に設定します。

デフォルト: `true`

`namedShadows`

(オプション) 同期する名前付きコアデバイスシャドウのリスト。シャドウの正確な名前を指定する必要があります。

> **Warning**
>
> AWS IoT Greengrass サービス
> は、`AWSManagedGreengrassV2Deployment`名前付きシャドウを使用して、個々のコアデバイスを対象とするデプロイを管理します。この名前付きシャドウは、 AWS IoT Greengrass サービスで使用するために予約されています。この名前付きシャドウを更新または削除しないでください。

`shadowDocumentsMap`

(オプション) 同期する追加のデバイスシャドウ。この構成パラメータを使用すると、シャドウドキュメントの指定が簡単になります。`shadowDocuments` オブジェクトの代わりに、このパラメータを使用することをお勧めします。

> **Note**
>
> `shadowDocumentsMap` オブジェクトを指定する場合は、`shadowDocuments` オブジェクトを指定しないでください。

各オブジェクトには、次の情報が含まれます:

`thingName`

このシャドウ構成の *`[thingName]`* のシャドウ設定。

`classic`

(オプション) `thingName` デバイスのクラシックデバイスシャドウを同期しない場合は、これを `false` に設定します。

`namedShadows`

同期する名前付きシャドウのリスト。シャドウの正確な名前を指定する必要があります。

`shadowDocuments`

(オプション) 同期する追加のデバイスシャドウのリスト。代わりに `shadowDocumentsMap` パラメータを使用することをお勧めします。

> **Note**
>
> `shadowDocuments` オブジェクトを指定する場合は、`shadowDocumentsMap` オブジェクトを指定しないでください。

このリストの各オブジェクトには、次の情報が含まれます。

`thingName`

シャドウを同期するデバイスのモノの名前。

`classic`

(オプション) `thingName` デバイスのクラシックデバイスシャドウを同期しない場合は、これを `false` に設定します。

デフォルト: `true`

`namedShadows`

(オプション) 同期する名前付きデバイスシャドウのリスト。シャドウの正確な名前を指定する必要があります。

`rateLimits`

(オプション) シャドウサービス要求のレート制限を決定する設定。

このオブジェクトには、次の情報が含まれます。

`maxOutboundSyncUpdatesPerSecond`

(オプション) デバイスが送信する 1 秒あたりの同期要求の最大数。

デフォルト: 100 要求/秒

`maxTotalLocalRequestsRate`

(オプション) コアデバイスに送信される 1 秒あたりのローカル IPC 要求の最大数。

デフォルト: 200 要求/秒

`maxLocalRequestsPerSecondPerThing`

(オプション) 接続された IoT モノごとに送信される 1 秒あたりのローカル IPC 要求の最大数。

デフォルト: 1 件につき 20 要求/秒

> **Note**
>
> これらのレート制限パラメータは、ローカルシャドウサービスの 1 秒あたりの最大要求数を定義します。 AWS IoT Device Shadow サービスの 1 秒あたりの最大リクエスト数は、 によって異なります AWS リージョン。詳しくは、「Amazon Web Services 全般のリファレンス」の「AWS IoT デバイスシャドウサービス API」の制限を参照ください。

`shadowDocumentSizeLimitBytes`

(オプション) ローカルシャドウの各 JSON 状態ドキュメントの最大許容サイズ。

この値を大きくすると、クラウドシャドウの JSON 状態ドキュメントのリソース制限も増やす必要があります。詳しくは、「Amazon Web Services 全般のリファレンス」の「AWS IoT デバイスシャドウサービス API」の制限を参照ください。

デフォルト: 8,192 バイト

最大: 30720 バイト

Example 例: 設定マージの更新

次の例は、シャドウマネージャーコンポーネントで利用可能なすべての設定パラメータを使用した設定マージ更新のサンプルを示しています。

```
{
   "strategy":{
      "type":"periodic",
      "delay":300
   },
   "synchronize":{
      "shadowDocumentsMap":{
         "MyDevice1":{
            "classic":false,
            "namedShadows":[
               "MyShadowA",
               "MyShadowB"
            ]
         },
         "MyDevice2":{
            "classic":true,
            "namedShadows":[]
         }
      },
      "direction":"betweenDeviceAndCloud"
   },
   "rateLimits":{
      "maxOutboundSyncUpdatesPerSecond":100,
      "maxTotalLocalRequestsRate":200,
      "maxLocalRequestsPerSecondPerThing":20
```

```
    },
    "shadowDocumentSizeLimitBytes":8192
}
```

2.0.x

`synchronize`

(オプション) シャドウを AWS クラウド と同期する方法を決定する同期設定。

> **Note**
>
> シャドウを AWS クラウド と同期させるには、このプロパティを使用して設定アップデートを作成する必要があります。

このオブジェクトには、次の情報が含まれます。

`coreThing`

(オプション) 同期するコアデバイスシャドウ。このオブジェクトには、次の情報が含まれます。

`classic`

(オプション) デフォルトでは、シャドウマネージャーはコアデバイスのクラシックシャドウのローカル状態を AWS クラウド と同期させます。クラシックデバイスシャドウを同期しない場合は、これを `false` に設定します。

デフォルト: `true`

`namedShadows`

(オプション) 同期する名前付きコアデバイスシャドウのリスト。シャドウの正確な名前を指定する必要があります。

> **Warning**
>
> AWS IoT Greengrass サービス
> は、`AWSManagedGreengrassV2Deployment`名前付きシャドウを使用して、個々のコアデバイスを対象とするデプロイを管理します。この名前付きシャド

ウは、 AWS IoT Greengrass サービスで使用するために予約されています。この名前付きシャドウを更新または削除しないでください。

`shadowDocumentsMap`

(オプション) 同期する追加のデバイスシャドウ。この構成パラメータを使用すると、シャドウドキュメントの指定が簡単になります。`shadowDocuments` オブジェクトの代わりに、このパラメータを使用することをお勧めします。

> **Note**
>
> `shadowDocumentsMap` オブジェクトを指定する場合は、`shadowDocuments` オブジェクトを指定しないでください。

各オブジェクトには、次の情報が含まれます:

`thingName`

このシャドウ構成の *`[thingName]`* のシャドウ設定。

`classic`

(オプション) `thingName` デバイスのクラシックデバイスシャドウを同期しない場合は、これを `false` に設定します。

`namedShadows`

同期する名前付きシャドウのリスト。シャドウの正確な名前を指定する必要があります。

`shadowDocuments`

(オプション) 同期する追加のデバイスシャドウのリスト。代わりに `shadowDocumentsMap` パラメータを使用することをお勧めします。

> **Note**
>
> `shadowDocuments` オブジェクトを指定する場合は、`shadowDocumentsMap` オブジェクトを指定しないでください。

このリストの各オブジェクトには、次の情報が含まれます。

`thingName`

シャドウを同期するデバイスのモノの名前。

`classic`

(オプション) `thingName` デバイスのクラシックデバイスシャドウを同期しない場合は、これを `false` に設定します。

デフォルト: `true`

`namedShadows`

(オプション) 同期する名前付きデバイスシャドウのリスト。シャドウの正確な名前を指定する必要があります。

`rateLimits`

(オプション) シャドウサービス要求のレート制限を決定する設定。

このオブジェクトには、次の情報が含まれます。

`maxOutboundSyncUpdatesPerSecond`

(オプション) デバイスが送信する 1 秒あたりの同期要求の最大数。

デフォルト: 100 要求/秒

`maxTotalLocalRequestsRate`

(オプション) コアデバイスに送信される 1 秒あたりのローカル IPC 要求の最大数。

デフォルト: 200 要求/秒

`maxLocalRequestsPerSecondPerThing`

(オプション) 接続された IoT モノごとに送信される 1 秒あたりのローカル IPC 要求の最大数。

デフォルト: 1 件につき 20 要求/秒

> **Note**
>
> これらのレート制限パラメータは、ローカルシャドウサービスの 1 秒あたりの最大要求数を定義します。 AWS IoT Device Shadow サービスの 1 秒あたりの最大リクエス

ト数は、 によって異なります AWS リージョン。詳しくは、「Amazon Web Services 全般のリファレンス」の「AWS IoT デバイスシャドウサービス API」の制限を参照ください。

`shadowDocumentSizeLimitBytes`

(オプション) ローカルシャドウの各 JSON 状態ドキュメントの最大許容サイズ。

この値を大きくすると、クラウドシャドウの JSON 状態ドキュメントのリソース制限も増やす必要があります。詳しくは、「Amazon Web Services 全般のリファレンス」の「AWS IoT デバイスシャドウサービス API」の制限を参照ください。

デフォルト: 8,192 バイト

最大: 30720 バイト

Example 例: 設定マージの更新

次の例は、シャドウマネージャーコンポーネントで利用可能なすべての設定パラメータを使用した設定マージ更新のサンプルを示しています。

```
{
  "synchronize": {
    "coreThing": {
      "classic": true,
      "namedShadows": [
        "MyCoreShadowA",
        "MyCoreShadowB"
      ]
    },
    "shadowDocuments": [
      {
        "thingName": "MyDevice1",
        "classic": false,
        "namedShadows": [
          "MyShadowA",
          "MyShadowB"
        ]
      },
      {
```

```
        "thingName": "MyDevice2",
        "classic": true,
        "namedShadows": []
      }
    ]
  },
  "rateLimits": {
    "maxOutboundSyncUpdatesPerSecond": 100,
    "maxTotalLocalRequestsRate": 200,
    "maxLocalRequestsPerSecondPerThing": 20
  },
  "shadowDocumentSizeLimitBytes": 8192
}
```

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* を AWS IoT Greengrass ルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.3.6	バグ修正と機能向上 • デバイスのオフライン中に AWS クラウド 更新によって削除されたシャドウプロパティが、接続の再取得後もローカルシャドウに引き続き存在する問題を修正しました。
2.3.5	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.3.4	バグ修正と機能向上 • null および空のシャドウ状態ドキュメントのサポートが追加されました。
2.3.3	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.3.2	バグ修正と機能向上 • ローカルシャドウデータベースが破損している場合に、シャドウマネージャーが BROKEN 状態になる問題を修正します。 • Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.3.1	バグ修正と機能向上 • クラウドシャドウアップデートの同期を妨げる可能性のある状態が修正されています。 • 名前付きシャドウ同期設定の変更が 1 つの名前付きシャドウにのみ適用される問題が修正されています。
2.3.0	バグ修正と機能向上 • Greengrass デバイスのプライベートキーがハードウェアセキュリティモジュールに保存されている場合にシャドウが同期されない可能性がある問題を修正しました。

バージョン	変更
2.2.4	バグ修正と機能向上 • ローカルのシャドウドキュメントを更新するときに、シャドウのサイズの検証がクラウドと一貫しない問題を修正しました。 • デプロイが設定ノードに対して RESET を実行する場合に、シャドウマネージャーが設定の更新のリッスンを停止する問題を修正しました。
2.2.3	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.2.2	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.2.1	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.2.0	新機能 • ローカルの公開/サブスクライブインターフェイスを通じた、ローカルシャドウサービスに関するサポートが追加されています。シャドウ MQTT トピックで、ローカルの公開/サブスクライブのメッセージブローカーと通信し、コアデバイスのシャドウを取得、更新、削除することが可能です。この機能により、MQTT ブリッジを使用してクライアントデバイスをローカルシャドウサービスに接続することで、シャドウトピックのメッセージを、クライアントデバイスとローカルの公開/サブスクライブインターフェイス間でリレーすることが可能になります。 この機能には、Greengrass nucleus コンポーネントの v2.6.0 以降が必要です。クライアントデバイスをローカルシャドウサービスに接続するには、MQTT ブリッジコンポーネント の v2.2.0 以降を使用する必要があります。 • ローカルシャドウサービスと AWS クラウド の間で、シャドウを同期する方向をカスタマイズするために、`direction` オプションを設定できるようになりました。このオプションを設定すると、 AWS クラウドへの帯域幅と接続数を低減できます。

バージョン	変更
2.1.1	バグ修正と機能向上 • JSON デバイスシャドウ状態ドキュメントの `desired` セクションと `reported` セクションの最大深度が5レベルではなく4レベルであった問題を修正します。 • Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.0	新機能 • 定期的なシャドウ同期間隔のサポートが追加されるため、帯域幅の使用量と料金を削減するようにコアデバイスを設定できます。
2.0.6	このバージョンには、バグ修正と機能向上が含まれています。
2.0.5	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.0.4	バグ修正と機能向上 • シャドウマネージャーが以前に削除されたシャドウの新しく作成されたバージョンを削除する問題が修正されます。 • `DeleteThingShadow` IPC 操作を更新して、呼び出されたときにシャドウバージョンをインクリメントします。
2.0.3	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.0.2	バグ修正と機能向上 • AWS IoT Core からシャドウ状態を同期するときにシャドウマネージャーが `delta` プロパティを認識しない問題を修正しました。 • シャドウの同期要求が誤ってマージされることがある問題を修正しました。
2.0.1	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.0.0	当初のバージョン

Amazon SNS

Amazon SNS コンポーネント (`aws.greengrass.SNS`) は、Amazon Simple Notification Service (Amazon SNS) トピックにメッセージを公開します。このコンポーネントを使用して、Greengrass コアデバイスから Web サーバー、E メールアドレス、その他のメッセージサブスクライバーにイベントを送信できます。詳細については、「Amazon Simple Notification Service デベロッパーガイド」の「Amazon SNS とは」を参照してください。

このコンポーネントを使用して Amazon SNS トピックに公開するには、このコンポーネントがサブスクライブしているトピックにメッセージを公開します。デフォルトでは、このコンポーネントは `sns/message` ローカルパブリッシュ/サブスクライブトピックにサブスクライブします。このコンポーネントをデプロイするときに、 AWS IoT Core MQTT トピックを含む他のトピックを指定できます。

カスタムコンポーネントでは、このコンポーネントに公開する前に、他のソースからのメッセージを処理するために、フィルタリングまたは書式設定ロジックを実装することができます。これにより、メッセージ処理ロジックを 1 つのコンポーネントに一元化できます。

> **Note**
>
> このコンポーネントは、 AWS IoT Greengrass V1 の Amazon SNS コネクタと同様の機能を提供します。詳細については、「AWS IoT Greengrass デベロッパーガイド」の「[Amazon SNS connector] (Amazon SNS コネクタ)」を参照してください。

トピック

- 変更ログ

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x
- 2.0.x

タイプ

このコンポーネントは Lambda コンポーネントです (`aws.greengrass.lambda`)。Greengrass nucleus は、Lambda ランチャーコンポーネントを使用してこのコンポーネントの Lambda 関数を実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、Linux コアデバイスにのみインストールできます。

要件

このコンポーネントには次の要件があります。

- コアデバイスは、Lambda 関数を実行するための要件を満たしている必要があります。コアデバイスが、コンテナ化された Lambda 関数を実行させる場合、そのデバイスは要件を満たす必要があります。詳細については、「Lambda 関数の要件」を参照してください。
- Python バージョン 3.7 がコアデバイスにインストールされ、PATH 環境変数に追加されています。
- Amazon SNS トピック 詳細については、Amazon Simple 通知サービス デベロッパーガイドの「Amazon SNS トピックの作成」を参照してください。
- 次の IAM ポリシーの例で示されているように、Greengrass デバイスのロールは `sns:Publish` アクションを許可する必要があります。

```
{
  "Version": "2012-10-17",
  "Statement": [
```

```
    {
      "Action": [
        "sns:Publish"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:sns:region:account-id:topic-name"
      ]
    }
  ]
}
```

このコンポーネントの入力メッセージペイロードのデフォルトトピックを動的にオーバーライドできます。アプリケーションでこの機能を使用する場合、IAM ポリシーにはすべてのターゲットトピックをリソースとして含める必要があります。リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (たとえば、ワイルドカード * 命名スキームを使用)。

- このコンポーネントから出力データを受信するには、このコンポーネントをデプロイするときに、次の設定更新プログラムをレガシーサブスクリプションルーターのコンポーネント (aws.greengrass.LegacySubscriptionRouter) のためにマージする必要があります。この設定は、このコンポーネントがレスポンスを公開するトピックを指定します。

  Legacy subscription router v2.1.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-sns": {
        "id": "aws-greengrass-sns",
        "source": "component:aws.greengrass.SNS",
        "subject": "sns/message/status",
        "target": "cloud"
      }
    }
  }
  ```

  Legacy subscription router v2.0.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-sns": {
        "id": "aws-greengrass-sns",
        "source": "arn:aws:lambda:region:aws:function:aws-greengrass-sns:version",
  ```

```
      "subject": "sns/message/status",
      "target": "cloud"
    }
  }
}
```

- *region* AWS リージョン は、使用する に置き換えます。
- *#####*を、このコンポーネントが実行する Lambda 関数のバージョンに置き換えます。Lambda 関数のバージョンを確認するには、デプロイするこのコンポーネントのバージョンの recipe を確認する必要があります。AWS IoT Greengrass コンソールで、このコンポーネントの詳細ページを開き、[Lambda function] (Lambda 関数) の key-value ペアを見つけます。このキー値のペアには、Lambda 関数の名前とバージョンが含まれます。

> **Important**
>
> このコンポーネントをデプロイするたびに、レガシーサブスクリプションルーターの Lambda 関数のバージョンを更新する必要があります。これにより、デプロイするコンポーネントバージョンに正しい Lambda 関数のバージョンが使用されることが保証されます。

詳細については、「デプロイの作成」を参照してください。

- Amazon SNS コンポーネントは、VPC での実行がサポートされています。このコンポーネントを VPC にデプロイするには、以下が必要です。
  - Amazon SNS コンポーネントには、VPC エンドポイントが `sns.region.amazonaws.com`である への接続が必要ですcom.amazonaws.us-east-1.sns。

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
sns.*region*.amazonaws.com	443	はい	Amazon SNS に

エンドポイント	ポート	必要	説明
			メッセージを公開します。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.7

次の表に、このコンポーネントのバージョン 2.1.7 に関する依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.6

次の表に、このコンポーネントのバージョン 2.1.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ハード

依存関係	互換性のあるバージョン	依存関係タイプ
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.5

次の表に、このコンポーネントのバージョン 2.1.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.4

次の表に、このコンポーネントのバージョン 2.1.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.3

次の表に、このコンポーネントのバージョン 2.1.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.2

次の表に、このコンポーネントのバージョン 2.1.2 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.8 - 2.1.0

次の表に、このコンポーネントのバージョン 2.0.8 および 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.7

次の表に、このコンポーネントのバージョン 2.0.7 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.6

次の表に、このコンポーネントのバージョン 2.0.6 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト

依存関係	互換性のあるバージョン	依存関係タイプ
トークン交換サービス	^2.0.0	ハード

2.0.5

次の表に、このコンポーネントのバージョン 2.0.5 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.4

次の表に、このコンポーネントのバージョン 2.0.4 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ハード
Lambda ランチャー	^2.0.0	ハード
Lambda ランタイム	^2.0.0	ソフト
トークン交換サービス	^2.0.0	ハード

2.0.3

次の表に、このコンポーネントのバージョン 2.0.3 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.3 <2.1.0	ハード
Lambda ランチャー	>=1.0.0	ハード
Lambda ランタイム	>=1.0.0	ソフト
トークン交換サービス	>=1.0.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

> **Note**
>
> このコンポーネントのデフォルト設定には、Lambda 関数のパラメータが含まれます。デバイスにこのコンポーネントを設定するには、次のパラメータのみを編集することをお勧めします。

`lambdaParams`

このコンポーネントの Lambda 関数のパラメータを含むオブジェクト。このオブジェクトには、次の情報が含まれます。

`EnvironmentVariables`

Lambda 関数のパラメータを含むオブジェクト。このオブジェクトには、次の情報が含まれます。

`DEFAULT_SNS_ARN`

このコンポーネントがメッセージを公開するデフォルトの Amazon SNS トピックの ARN。入力メッセージペイロードの `sns_topic_arn` プロパティを使用して、宛先トピックをオーバーライドできます。

containerMode

(オプション) このコンポーネントのコンテナ化モード。次のオプションから選択します。

- NoContainer - コンポーネントは、分離されたランタイム環境では実行されません。
- GreengrassContainer – コンポーネントは、 AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。

デフォルト: GreengrassContainer

containerParams

(オプション) このコンポーネントのコンテナパラメータを含むオブジェクト。containerMode の GreengrassContainer を指定した場合、コンポーネントはこれらのパラメータを使用します。

このオブジェクトには、次の情報が含まれます。

memorySize

(オプション) コンポーネントに割り当てるメモリ量 (KB 単位)。

デフォルトは 512 MB (525,312 KB) です。

pubsubTopics

(オプション) コンポーネントがメッセージを受信するためにサブスクライブするトピックを含むオブジェクト。各トピックと、コンポーネントが から MQTT トピックをサブスクライブするか、ローカルのパブリッシュ/サブスクライブトピックをサブスクライブ AWS IoT Core するかを指定できます。

このオブジェクトには、次の情報が含まれます。

0 - これは文字列としての配列インデックスです。

次の情報が含まれるオブジェクト。

type

(オプション) このコンポーネントがメッセージをサブスクライブするために使用するパブリッシュ/サブスクライブメッセージングのタイプ。次のオプションから選択します。

- PUB_SUB - ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブします。このオプションを選択した場合、トピックに MQTT ワイルドカードを含めることはできません。このオプションを指定したときに、カスタムコンポーネントからメッセー

ジを送信する方法の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。

- IOT_CORE – AWS IoT Core MQTT メッセージをサブスクライブします。このオプションを選択した場合、トピックに MQTT ワイルドカードを含めることができます。このオプションを指定したときに、カスタムコンポーネントからメッセージを送信する方法の詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。

デフォルト: PUB_SUB

`topic`

(オプション) コンポーネントがメッセージを受信するためにサブスクライブするトピック。type の IotCore を指定した場合、このトピックで MQTT ワイルドカード (+ および #) を使用できます。

Example 例: 設定マージの更新 (コンテナモード)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "DEFAULT_SNS_ARN": "arn:aws:sns:us-west-2:123456789012:mytopic"
    }
  },
  "containerMode": "GreengrassContainer"
}
```

Example 例: 設定マージの更新 (コンテナモードなし)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "DEFAULT_SNS_ARN": "arn:aws:sns:us-west-2:123456789012:mytopic"
    }
  },
  "containerMode": "NoContainer"
}
```

入力データ

このコンポーネントは、次のトピックに関するメッセージを受け取り、その情報をそのままターゲット Amazon SNS トピックに公開します。デフォルトで、このコンポーネントはローカルのパブリッシュ/サブスクライブメッセージにサブスクライブします。カスタムコンポーネントからこのコンポーネントにメッセージをパブリッシュする方法の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。

デフォルトトピック (ローカルパブリッシュ/サブスクライブ): `sns/message`

メッセージは、次のプロパティを受付けます。入力メッセージは JSON 形式である必要があります。

`request`

Amazon SNS トピックに送信するメッセージに関する情報。

タイプ: 次の情報が含まれる `object`。

`message`

文字列としてのメッセージの内容。

JSON オブジェクトを送信するには、JSON オブジェクトを文字列としてシリアル化し、`message_structure` プロパティに `json` を指定します。

タイプ: `string`

`subject`

(オプション) メッセージの件名

タイプ: `string`

件名は ASCII テキストで、最大 100 文字です。文字、数字、または句読点で始まる必要があります。改行や制御文字を含めることはできません。

`sns_topic_arn`

(オプション) このコンポーネントがメッセージを公開する Amazon SNSトピックの ARN。デフォルトの Amazon SNS トピックをオーバーライドするには、このプロパティを指定します。

タイプ: `string`

`message_structure`

(オプション) メッセージの構造。`content` プロパティで文字列としてシリアル化する JSON メッセージを送信するには、`json` を指定します。

タイプ: `string`

有効な値: `json`

`id`

リクエストの任意の ID。このプロパティを使用して、入力リクエストを出力レスポンスにマッピングします。このプロパティを指定するとき、コンポーネントはこの値に対してレスポンスオブジェクトの `id` プロパティを設定します。

タイプ: `string`

> **Note**
>
> メッセージサイズは最大 256 KB です。

Example 入力例: 文字列メッセージ

```
{
  "request": {
    "subject": "Message subject",
    "message": "Message data",
    "sns_topic_arn": "arn:aws:sns:region:account-id:topic2-name"
  },
  "id": "request123"
}
```

Example 入力例: JSON メッセージ

```
{
  "request": {
    "subject": "Message subject",
    "message": "{ \"default\": \"Message data\" }",
    "message_structure": "json"
```

```
  },
  "id": "request123"
}
```

出力データ

このコンポーネントは、デフォルトで次の MQTT トピックに出力データとしてレスポンスを公開します。このトピックは、[legacy subscription router component] (レガシーサブスクリプションルーターコンポーネント) の設定で `subject` として指定する必要があります。カスタムコンポーネントでこのトピックに関するメッセージへサブスクライブする方法の詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。

デフォルトトピック (AWS IoT Core MQTT): `sns/message/status`

Example 出力例: 成功

```
{
  "response": {
    "sns_message_id": "f80a81bc-f44c-56f2-a0f0-d5af6a727c8a",
    "status": "success"
  },
  "id": "request123"
}
```

Example 出力例: 失敗

```
{
  "response" : {
    "error": "InvalidInputException",
    "error_message": "SNS Topic Arn is invalid",
    "status": "fail"
  },
  "id": "request123"
}
```

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

```
/greengrass/v2/logs/aws.greengrass.SNS.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。を AWS IoT Greengrass ルートフォルダへのパス*/greengrass/v2*に置き換えます。

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.SNS.log
```

ライセンス

このコンポーネントには、次のサードパーティーソフトウェア/ライセンス品が含まれています。

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF ライセンス
- docutils/BSD ライセンス、GNU 一般パブリックライセンス (GPL)、Python Software Foundation ライセンス、パブリックドメイン
- jmespath/MIT ライセンス
- s3transfer/Apache License 2.0
- urllib3/MIT ライセンス

このコンポーネントは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされています。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.7	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.1.6	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.1.5	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.4	Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.3	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.2	Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.1.1	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.1.0	新機能 • HTTPS ネットワークプロキシ設定へのサポートが追加されました。詳細については、ポート 443 での接続またはネットワークプロキシを通じた接続およびコアデバイスが HTTPS プロキシを信頼できるようにするを参照してください。
2.0.8	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。
2.0.7	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.0.6	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.0.5	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.0.4	Greengrass nucleus バージョン 2.1.0 のリリース用にバージョンが更新されました。
2.0.3	当初のバージョン

ストリームマネージャー

ストリームマネージャーコンポーネント (`aws.greengrass.StreamManager`) を使用すると、データストリームを処理して AWS クラウド Greengrass コアデバイスから に転送できます。

カスタムコンポーネントでストリームマネージャーを設定して使用する方法の詳細については、「Greengrass コアデバイスでのデータストリームの管理」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.1.x
- 2.0.x

> **Note**
>
> ストリームマネージャーを使用してデータをクラウドにエクスポートする場合、ストリームマネージャーコンポーネントのバージョン 2.0.7 を v2.0.8 と v2.0.11 の間のバージョンにアップグレードすることはできません。ストリームマネージャーを初めてデプロイする場合、ストリームマネージャーコンポーネントの最新バージョンをデプロイすることを強くお勧めします。

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- トークン交換ロールは、ストリームマネージャーで使用する AWS クラウド 送信先へのアクセスを許可する必要があります。詳細については、以下を参照してください。
  - the section called “AWS IoT Analytics チャネル”
  - the section called “Amazon Kinesis Data Streams”
  - the section called “AWS IoT SiteWise アセットプロパティ”
  - the section called “Amazon S3 オブジェクト”
- ストリームマネージャーコンポーネントは、VPC での実行がサポートされています。このコンポーネントを VPC にデプロイするには、以下が必要です。
  - ストリームマネージャーコンポーネントには、データを発行する AWS サービスへの接続が必要です。
    - Amazon S3: `com.amazonaws.region.s3`
    - Amazon Kinesis Data Streams: `com.amazonaws.region.kinesis-streams`
    - AWS IoT SiteWise: `com.amazonaws.region.iotsitewise.data`
  - `us-east-1` リージョンの Amazon S3 にデータを発行する場合、このコンポーネントはデフォルトで S3 グローバルエンドポイントの使用を試みますが、このエンドポイントは Amazon S3 VPC インターフェイスエンドポイントからは使用できません。詳細については、「Amazon S3

の の制限と制限 AWS PrivateLink」を参照してください。これを解決するには、次のオプションから選択できます。

- AWS_S3_US_EAST_1_REGIONAL_ENDPOINT=regional 環境変数を指定して、us-east-1リージョンのリージョン S3 エンドポイントを使用するようにストリームマネージャーコンポーネントを設定します。
- Amazon S3 インターフェイス VPC エンドポイントの代わりに Amazon S3 ゲートウェイ VPC エンドポイントを作成します。S3 ゲートウェイエンドポイントは、S3 グローバルエンドポイントへのアクセスをサポートします。詳細については、「ゲートウェイエンドポイントの作成」を参照してください。

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
iotanalytics. *region*.amazonaws.com	443	いいえ	にデータを発行する場合は必須です AWS IoT Analytics。
kinesis.*region*.amazonaws.com	443	いいえ	Firehose にデータを発行する場合は必須です。
data.iotsitewise. *region*.amazonaws.com	443	いいえ	にデータを発行する場合は必須です AWS IoT SiteWise。

エンドポイント	ポート	必要	説明
*.s3.amazonaws.com	443	いいえ	S3 バケットにデータを公開する場合に必要です。 * は、データを公開する各バケットの名前に置き換えることができます。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

2.1.11

次の表に、このコンポーネントのバージョン 2.1.11 から 2.1.10 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.13.0	ソフト
トークン交換サービス	>=0.0.0	ハード

2.1.9 – 2.1.10

次の表に、このコンポーネントのバージョン 2.1.9 から 2.1.10 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.12.0	ソフト
トークン交換サービス	>=0.0.0	ハード

2.1.5 – 2.1.8

次の表に、このコンポーネントのバージョン 2.1.5 から 2.1.8 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.11.0	ソフト
トークン交換サービス	>=0.0.0	ハード

2.1.2 – 2.1.4

次の表に、このコンポーネントのバージョン 2.1.2 から 2.1.4 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.10.0	ソフト
トークン交換サービス	>=0.0.0	ハード

2.1.1

次の表に、このコンポーネントのバージョン 2.1.1 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.9.0	ソフト

依存関係	互換性のあるバージョン	依存関係タイプ
トークン交換サービス	>=0.0.0	ハード

2.1.0

次の表に、このコンポーネントのバージョン 2.1.0 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.8.0	ソフト
トークン交換サービス	>=0.0.0	ハード

2.0.15

次の表に、このコンポーネントのバージョン 2.0.15 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.7.0	ソフト
トークン交換サービス	>=0.0.0	ハード

2.0.13 and 2.0.14

次の表に、このコンポーネントのバージョン 2.0.13 および 2.0.14 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.6.0	ソフト
トークン交換サービス	>=0.0.0	ハード

2.0.11 and 2.0.12

次の表に、このコンポーネントのバージョン 2.0.11 および 2.0.12 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.5.0	ソフト
トークン交換サービス	>=0.0.0	ハード

2.0.10

次の表に、このコンポーネントのバージョン 2.0.10 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.4.0	ソフト
トークン交換サービス	>=0.0.0	ハード

2.0.9

次の表に、このコンポーネントのバージョン 2.0.9 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.3.0	ソフト
トークン交換サービス	>=0.0.0	ハード

2.0.8

次の表に、このコンポーネントのバージョン 2.0.8 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.0 <2.2.0	ソフト
トークン交換サービス	>=0.0.0	ハード

2.0.7

次の表に、このコンポーネントのバージョン 2.0.7 の依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.0.3 <2.1.0	ソフト
トークン交換サービス	>=0.0.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

`STREAM_MANAGER_STORE_ROOT_DIR`

(オプション) ストリームを保存するために使用されるローカルディレクトリの絶対パス。この値は、スラッシュ (`/data` など) で開始する必要があります。

既存のフォルダを指定する必要があります。ストリームマネージャーコンポーネントを実行するシステムユーザーには、このフォルダに対する読み取りと書き込み許可が必要です。例えば、次のコマンドを実行して、ストリームマネージャーのルートフォルダとして指定するフォルダ `/var/greengrass/streams` を作成および設定できます。これらのコマンドは、デフォルトのシステムユーザーである `ggc_user` が、このフォルダを読み取りおよび書き込みできるようにします。

```
sudo mkdir /var/greengrass/streams
sudo chown ggc_user /var/greengrass/streams
sudo chmod 700 /var/greengrass/streams
```

デフォルト: */greengrass/v2*/work/aws.greengrass.StreamManager

`STREAM_MANAGER_SERVER_PORT`

(オプション) ストリームマネージャーとの通信に使用するローカルポート番号。

`0` を指定して、ランダムに利用可能なポートを利用できます。

デフォルト: `8088`

`STREAM_MANAGER_AUTHENTICATE_CLIENT`

(オプション) クライアントがストリームマネージャーとやり取りする前に、クライアントの認証を必須にできます。ストリームマネージャー SDK は、クライアントとストリームマネージャー間の相互作用を制御します。このパラメータは、ストリームを操作するためにストリームマネージャー SDK を呼び出すことができるクライアントを決定します。詳細については、「ストリームマネージャークライアント認証」を参照してください。

`true` を指定した場合、ストリームマネージャー SDK では Greengrass コンポーネントのみをクライアントとして許可します。

`false` を指定した場合、ストリームマネージャー SDK では、コアデバイスのすべてのプロセスをクライアントにすることができます。

デフォルト: `true`

`STREAM_MANAGER_EXPORTER_MAX_BANDWIDTH`

(オプション) ストリームマネージャーがデータのエクスポートに使用できる平均最大帯域幅 (KB/s) 。

デフォルト: 無制限

`STREAM_MANAGER_EXPORTER_THREAD_POOL_SIZE`

(オプション) ストリームマネージャーがデータのエクスポートに使用できるアクティブなスレッドの最大数。

最適なサイズは、ハードウェア、ストリームボリューム、予定されているエクスポートストリームの数によって異なります。エクスポート速度が遅い場合は、この設定を調整して、ハードウェアとビジネスケースに最適なサイズを見つけることができます。コアデバイスハードウェアの CPU とメモリは、制限要因です。開始するには、この値をデバイスのプロセッサコアの数と同じ値に設定してみてください。

ハードウェアがサポートできるサイズよりも大きいサイズを設定しないように注意してください。各ストリームはハードウェアリソースを消費するため、制約のあるデバイス上ではエクスポートストリームの数を制限する必要があります。

デフォルト: 5 スレッド

`STREAM_MANAGER_EXPORTER_S3_DESTINATION_MULTIPART_UPLOAD_MIN_PART_SIZE_BYTES`

(オプション) Amazon S3 にマルチパートアップロードのパートの最小サイズ (バイト単位)。ストリームマネージャーはこの設定と入力ファイルのサイズを基に、マルチパート PUT リクエストのデータをバッチ処理する方法を決定します。

> **Note**
>
> ストリームマネージャーは、ストリーム `sizeThresholdForMultipartUploadBytes` プロパティを使用して、Amazon S3 にシングルまたはマルチパートのアップロードとしてエクスポートするかどうかを決定します。 AWS IoT Greengrass コンポーネントは、Amazon S3 にエクスポートするストリームを作成するときに、このしきい値を設定できます。

デフォルト: `5242880` (5 MB)。これも最小値です。

`LOG_LEVEL`

(オプション) コンポーネントのログレベル。こちらにレベル順に一覧表示されているログレベルから選択します。

- `TRACE`
- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

デフォルト: `INFO`

`JVM_ARGS`

(オプション) 起動時にストリームマネージャーに渡すカスタム Java 仮想マシン引数。複数の引数はスペースで区切ります。

このパラメータは、JVM で使用されるデフォルト設定を上書きする必要がある場合にのみ使用します。例えば、大量のストリームをエクスポートする場合は、デフォルトのヒープサイズを大きくする必要があります。

Example 例: 設定マージの更新

次の設定例では、デフォルト以外のポートを使用するように指定しています。

```
{
  "STREAM_MANAGER_SERVER_PORT": "18088"
}
```

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/aws.greengrass.StreamManager.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.StreamManager.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* を AWS IoT Greengrass ルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.StreamManager.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.greengrass.StreamManager.log -Tail 10 -
  Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.1.12	バグ修正と機能向上 認証情報を使用する順序を更新して、 AWS サービスリクエストに Greengrass 認証情報が優先されるようにします。
2.1.11	Greengrass nucleus バージョン 2.12.0 リリース用にバージョンが更新されました。
2.1.10	バグ修正と機能向上 HTTPS プロキシ設定が Greengrass 認証機関 (CA) 証明書チェーンを信頼しない問題を修正しました。
2.1.9	Greengrass nucleus バージョン 2.11.0 のリリース用にバージョンが更新されました。
2.1.8	バグ修正と機能向上 ストリームマネージャーが で失敗した SiteWise エクスポートを無限に再試行する問題を修正しましたInvalidRequestException 。
2.1.7	バグ修正と機能向上 ストリームマネージャーがプロキシ構成を正しく読み取れない問題を修正しました。
2.1.6	バグ修正と機能向上 Jetson Nano など、特定の ARMv8 プロセッサで起動時にクラッシュする可能性がある問題を修正しました。
2.1.5	Greengrass nucleus バージョン 2.10.0 のリリース用にバージョンが更新されました。
2.1.4	バグ修正と機能向上 • 1 つのバッチ内で同じタイムスタンプを持つ同じプロパティアセットのエントリが SiteWise API ConflictingOperationException か

バージョン	変更
	ら返され、ストリームマネージャーが継続的に再試行する問題を修正しました。 • デフォルトの接続タイムアウトを 3 秒から 1 分に更新しました。
2.1.3	バグ修正と機能向上 システムユーザーとして Windows OS を実行する場合に発生する、起動時の問題を修正しました。
2.1.2	バグ修正と機能向上 • 英語以外の言語を使用する Windows OS における問題を修正しました。 • Greengrass nucleus バージョン 2.9.0 のリリース用にバージョンが更新されました。
2.1.1	Greengrass nucleus バージョン 2.8.0 のリリース用にバージョンが更新されました。
2.1.0	新機能 • このコンポーネントを更新して、テレメトリメトリクスを Amazon に自動的に送信します EventBridge。詳細については、「AWS IoT Greengrass コアデバイスからシステムヘルステレメトリデータを収集する」を参照してください。 この機能を使用するには、Greengrass nucleus コンポーネントの v2.7.0 以降が必要です。 • Greengrass nucleus バージョン 2.7.0 のリリース用にバージョンが更新されました。
2.0.15	Greengrass nucleus バージョン 2.6.0 のリリース用にバージョンが更新されました。
2.0.14	このバージョンには、バグ修正と機能向上が含まれています。
2.0.13	Greengrass nucleus バージョン 2.5.0 のリリース用にバージョンが更新されました。

バージョン	変更
2.0.12	バグ修正と機能向上 ストリームマネージャー v2.0.7 を v2.0.8 と v2.0.11 の間のバージョンにアップグレードできない問題を修正しました。ストリームマネージャーを使用してデータをクラウドにエクスポートする場合、v2.0.12 にアップグレードできるようになりました。
2.0.11	Greengrass nucleus バージョン 2.4.0 のリリース用にバージョンが更新されました。
2.0.10	Greengrass nucleus バージョン 2.3.0 のリリース用にバージョンが更新されました。
2.0.9	Greengrass nucleus バージョン 2.2.0 のリリース用にバージョンが更新されました。
2.0.8	Greengrass nucleus バージョン 2.1.0 のリリース用にバージョンが更新されました。
2.0.7	当初のバージョン

Systems Manager エージェント

AWS Systems Manager エージェントコンポーネント (`aws.greengrass.SystemsManagerAgent`) は Systems Manager エージェントをインストールするため、Systems Manager でコアデバイスを管理できます。Systems Manager は、Amazon EC2 インスタンス AWS、オンプレミスサーバー、仮想マシン (VM)、エッジデバイスなど、 上のインフラストラクチャを表示および制御するために使用できる AWS のサービスです。 VMs Systems Manager は、運用データの表示、運用タスクの自動化、セキュリティとコンプライアンスの維持を可能にします。詳細については、「 AWS Systems Manager ユーザーガイド」の「 とは AWS Systems Manager」および「Systems Manager エージェントについて」を参照してください。

Systems Manager のツールや特徴は、機能と呼ばれます。Greengrass コアデバイスは、すべての Systems Manager 機能をサポートしています。これらの機能と Systems Manager を使用してコアデバイスを管理する方法の詳細については、「AWS Systems Manager ユーザーガイド」の「Systems Manager 機能」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 1.1.x
- 1.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、Linux コアデバイスにのみインストールできます。

要件

このコンポーネントには次の要件があります。

- 64 ビット Linux プラットフォームで実行される Greengrass コアデバイス: Armv8 (AArch64) または x86_64。
- Systems Manager が引き受けることができる AWS Identity and Access Management (IAM) サービスロールが必要です。このロールには、AmazonSSMManagedInstanceCore 管理ポリシー、また

は同等のアクセス許可を定義するカスタムポリシーを含める必要があります。詳細については、「AWS Systems Manager ユーザーガイド」の「エッジデバイス用の IAM サービスロールを作成」を参照してください。

このコンポーネントをデプロイするときは、SSMRegistrationRole 設定パラメータにこのロールの名前を指定する必要があります。

- [Greengrass device role] (Greengrass デバイスロール)は、ssm:AddTagsToResource および ssm:RegisterManagedInstance アクションを許可する必要があります。デバイスロールは、前の要件を満たす IAM サービスロールの iam:PassRole アクションも許可する必要があります。以下の IAM ポリシーの例は、次の権限を付与します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "iam:PassRole"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:iam::account-id:role/SSMServiceRole"
      ]
    },
    {
      "Action": [
        "ssm:AddTagsToResource",
        "ssm:RegisterManagedInstance"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
ec2messages. *region*.amazonaws.com	443	はい	AWS クラウドの Systems Manager サービスと通信します。
ssm.*region*.amazonaws.com	443	はい	コアデバイスを Systems Manager マネージドノードとして登録します。
ssmmessages. *region*.amazonaws.com	443	はい	AWS クラウドで、Systems Manager の機能であるセッションマネージャーと通信します。

詳細については、「AWS Systems Manager ユーザーガイド」の「[Reference: ec2messages, ssmmessages, and other API calls] (リファレンス: ec2messages、ssmmessages と他の API コール)」を参照してください。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

次の表に、このコンポーネントのバージョン 1.0.0 から 1.1.0 までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
トークン交換サービス	^2.0.0	ソフト

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントには、コンポーネントのデプロイ時にカスタマイズできる次の設定パラメータが用意されています。

`SSMRegistrationRole`

Systems Manager が引き受けることができる IAM サービスロール。これには、AmazonSSMManagedInstanceCore 管理ポリシーまたは同等のアクセス許可を定義するカスタムポリシーが含まれます。詳細については、「AWS Systems Manager ユーザーガイド」の「エッジデバイス用の IAM サービスロールを作成」を参照してください。

`SSMOverrideExistingRegistration`

(オプション) コアデバイスがすでにハイブリッドアクティベーションで登録された Systems Manager エージェントを実行している場合は、デバイスの既存の Systems Manager エージェント登録をオーバーライドできます。このオプションを `true` に設定すると、このコンポーネントが提供する Systems Manager エージェントを使用して、コアデバイスをマネージドノードとして登録できます。

> **Note**
>
> このオプションは、ハイブリッドアクティベーションに登録されているデバイスにのみ適用されます。コアデバイスが Systems Manager エージェントがインストールされ、インスタンスプロファイルロールが設定された Amazon EC2 インスタンスで実行される場合、Amazon EC2 インスタンスの既存のマネージドノード ID は `i-` で始まります。Systems Manager エージェントコンポーネントをインストールすると、Systems Manager エージェントは、ID が `i-` ではなく `mi-` で始まる新しいマネージドノードを登録します。次に、ID が `mi-` で始まるマネージドノードを使用して、Systems Manager でコアデバイスを管理できます。

デフォルト: `false`

`SSMResourceTags`

(オプション) このコンポーネントがコアデバイス用に作成する Systems Manager マネージドノードに追加するタグ。これらのタグを使用して、Systems Manager でコアデバイスのグループを管理できます。たとえば、指定したタグを持つすべてのデバイスでコマンドを実行できます。

各タグが `Key` と `Value` のオブジェクトであるリストを指定します。例えば、次の `SSMResourceTags` の値は、コアデバイスのマネージドノードで **`Owner`** タグを **`richard-roe`** に設定するようにこのコンポーネントに指示します。

```
[
  {
    "Key": "Owner",
    "Value": "richard-roe"
  }
]
```

マネージドノードがすでに存在し、`SSMOverrideExistingRegistration` が `false` の場合、このコンポーネントはこれらのタグを無視します。

Example 例: 設定マージの更新

次の設定例では、`SSMServiceRole` という名前のサービスロールを使用して、コアデバイスが Systems Manager に登録して通信できるようにすることを指定しています。

```
{
```

```
  "SSMRegistrationRole": "SSMServiceRole",
  "SSMOverrideExistingRegistration": false,
  "SSMResourceTags": [
    {
      "Key": "Owner",
      "Value": "richard-roe"
    },
    {
      "Key": "Team",
      "Value": "solar"
    }
  ]
}
```

ローカルログファイル

Systems Manager エージェントソフトウェアは、Greengrass ルートフォルダの外部のフォルダにログを書き込みます。詳細については、「AWS Systems Manager ユーザーガイド」の「Systems Manager エージェントのログの表示」を参照してください。

Systems Manager エージェントコンポーネントは、シェルスクリプトを使用して Systems Manager エージェントをインストール、起動、および停止します。これらのスクリプトからの出力は、次のログファイルにあります。

```
/greengrass/v2/logs/aws.greengrass.SystemsManagerAgent.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。を AWS IoT Greengrass ルートフォルダへのパス*/greengrass/v2*に置き換えます。

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.SystemsManagerAgent.log
  ```

以下も参照してください。

- AWS Systems Manager で Greengrass コアデバイスを管理する
- 「 AWS Systems Managerユーザーガイド」の「AWS Systems Manager とは」

- 「AWS Systems Manager ユーザーガイド」の「[About Systems Manager Agent] (Systems Manager エージェントについて)」

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
1.1.0	このバージョンには、バグ修正と機能向上が含まれています。
1.0.0	当初のバージョン

トークン交換サービス

トークン交換サービスコンポーネント (`aws.greengrass.TokenExchangeService`) は、カスタムコンポーネントの AWS サービスとやり取りするために使用できる AWS 認証情報を提供します。

トークン交換サービスは、Amazon Elastic Container Service (Amazon ECS) コンテナインスタンスをローカルサーバーとして実行します。このローカルサーバーは、[Greengrass core nucleus component] (Greengrass core nucleus コンポーネント) で設定した AWS IoT ロールエイリアスを使用して、AWS IoT 認証情報プロバイダに接続します。このコンポーネントは、2 つの環境変数 `AWS_CONTAINER_CREDENTIALS_FULL_URI` および `AWS_CONTAINER_AUTHORIZATION_TOKEN` を提供します。`AWS_CONTAINER_CREDENTIALS_FULL_URI` はこのローカルサーバーへの URI を定義します。コンポーネントが AWS SDK クライアントを作成すると、クライアントはこの URI 環境変数を認識し、`AWS_CONTAINER_AUTHORIZATION_TOKEN` のトークンを使用してトークン交換サービスに接続し、AWS 認証情報を取得します。これにより、Greengrass コアデバイスは AWS サービスオペレーションを呼び出すことができます。カスタムコンポーネントでこのコンポーネントを使用する方法の詳細については、「AWS サービスとやり取り」を参照してください。

> **Important**
>
> この方法で AWS 認証情報を取得するサポートは、2016 年 7 月 13 日に AWS の SDK に追加されました。コンポーネントは、その日以降に作成された AWS SDK バージョンを使用する必要があります。詳細については、「Amazon Elastic Container Service デベロッパーガイド」の「サポートされる AWS SDK の使用」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

依存関係

このコンポーネントに依存関係はありません。

構成

このコンポーネントに設定パラメータはありません。

ローカルログファイル

このコンポーネントは、Greengrass nucleus コンポーネントと同じログファイルを使用します。

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.0.3	当初のバージョン

IoT SiteWise OPC-UA コレクター

IoT SiteWise OPC-UA コレクターコンポーネント (`aws.iot.SiteWiseEdgeCollectorOpcua`) を使用すると、 AWS IoT SiteWise ゲートウェイはローカル OPC-UA サーバーからデータを収集できます。

このコンポーネントを使用すると、 AWS IoT SiteWise ゲートウェイは複数の OPC-UA サーバーに接続できます。 AWS IoT SiteWise ゲートウェイの詳細については、「 AWS IoT SiteWise ユーザーガイド」の「エッジ AWS IoT SiteWise での の使用」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 2.4.x
- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Greengrass コアデバイスは、次のいずれかのプラットフォームで実行する必要があります。
  - OS: Ubuntu 18.04 以降

    アーキテクチャ: x86_64 (AMD64) または ARMv8 (Aarch64)
  - OS: Red Hat Enterprise Linux (RHEL) 8

    アーキテクチャ: x86_64 (AMD64) または ARMv8 (Aarch64)
  - OS: Amazon Linux 2

    アーキテクチャ: x86_64 (AMD64) または ARMv8 (Aarch64)
  - OS: Debian 11

    アーキテクチャ: x86_64 (AMD64) または ARMv8 (Aarch64)
  - OS: Windows Server 2019 以降

    アーキテクチャ: x86_64 (AMD64)
- Greengrass コアデバイスは、OPC-UA サーバーへのアウトバウンドネットワーク接続を許可する必要があります。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

次の表では、このコンポーネントのすべてのバージョンの依存関係を一覧表示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.3.0 <3.0.0	ハード
ストリームマネージャー	>2.0.10<3.0.0	ハード
シークレットマネージャー	>=2.0.8 <3.0.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントに設定パラメータはありません。

AWS IoT SiteWise コンソールまたは API を使用して IoT OPC-UA IoT SiteWise コレクターコンポーネントを設定できます。詳細については、「AWS IoT SiteWise ユーザーガイド」の「ステップ 4: データソースを追加する - オプション」を参照してください。

入力データ

このコンポーネントは、次の形式のデータのみを受け入れ、それ以外はすべて無視されて破棄されます。次の表は、OPC UA データ型を同等のデータ型にマッピングしています SiteWise。

SiteWise データ型	OPC UA データ型	説明
STRING	String Guid XmlElement	最大長 1024 バイトの文字列。
INTEGER	SByte Byte Int16 UInt16 Int32 UInt32* Int64*	からの範囲の符号付き 32 -2,147,483,648 to 2,147,483,647 ビット整数。
DOUBLE	UInt32* Int64* Float Double	の範囲が -10^100 to 10^100とIEEE 754倍精度の浮動小数点数。
BOOLEAN	Boolean	true-または-false

* OPC UA データ型 UInt32および ではInt64、 SiteWise がその値を表すことができるINTEGER場合、その SiteWise データ型は になります。それ以外の場合は になりますDOUBLE。

出力データ

このコンポーネントは、 AWS IoT Greengrass ストリームマネージャー にBatchPutAssetPropertyValueメッセージを書き込みます。詳細については、AWS IoT SiteWise API リファレンスの BatchPutAssetPropertyValue を参照してください。

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/aws.iot.SiteWiseEdgeCollectorOpcua.log
```

Windows

```
C:\greengrass\v2\logs\aws.iot.SiteWiseEdgeCollectorOpcua.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* を AWS IoT Greengrass ルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.iot.SiteWiseEdgeCollectorOpcua.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.iot.SiteWiseEdgeCollectorOpcua.log -Tail
   10 -Wait
  ```

トラブルシューティングとデバッグ

このコンポーネントには、お客様が問題を特定して解決するのに役立つ新しいイベントログが含まれています。ログファイルはローカルログファイルとは別のもので、次の場所にあります。*/greengrass/v2* または *C:\greengrass\v2* を AWS IoT Greengrass ルートフォルダへのパスに置き換えます。

Linux

```
/greengrass/v2/work/aws.iot.SiteWiseEdgeCollectorOpcua/logs/
IotSiteWiseOpcUaCollectorEvents.log
```

Windows

```
C:\greengrass\v2\work\aws.iot.SiteWiseEdgeCollectorOpcua\logs
\IotSiteWiseOpcUaCollectorEvents.log
```

このログには、詳細情報とトラブルシューティングの手順が含まれています。トラブルシューティングに関する情報は診断結果とともに提供され、問題の解決方法の説明と、場合によっては詳細情報へのリンクが含まれます。診断情報には次のものが含まれます。

- 重要度レベル
- タイムスタンプ
- イベント固有の追加情報

Example ログの例

```
dataSourceConnectionSuccess:
  Summary: Successfully connected to OpcUa server
  Level: INFO
  Timestamp: '2023-06-15T21:04:16.303Z'
  Description: Successfully connected to the data source.
  AssociatedMetrics:
  - Name: FetchedDataStreams
    Description: The number of fetched data streams for this data source
    Value: 1.0
    Namespace: IoTSiteWise
    Dimensions:
    - Name: SourceName
      Value: SourceName{value=OPC-UA Server}
    - Name: ThingName
      Value: test-core
  AssociatedData:
  - Name: DataSourceTrace
    Description: Name of the data source
    Data:
    - OPC-UA Server
  - Name: EndpointUri
    Description: The endpoint to which the connection was attempted.
    Data:
    - '"opc.tcp://10.0.0.1:1234"'
```

ライセンス

このコンポーネントは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされています。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
2.4.2	バグ修正と機能向上 • OPC UA サーバーの検出中にノードが複数回検出される可能性がある問題を修正しました。 • スナップショットデータポイントごとにタイムスタンプが新しくなるようにスナップショット機能を修正しました。
2.4.1	バグ修正と機能向上 • プロキシサポートに関連する問題を修正しました。 • スレッドのクリーンアップが失敗し、データブロックが発生する問題を修正しました。
2.4.0	新機能 • イベントログを追加して、問題の特定と修正を容易にしています。 バグ修正と機能向上 • OPC-UA の仕様のバージョン 1.05 を使用する OPC-UA サーバーに接続する際に証明書エラーを引き起こす OPC-UA クライアントの問題を修正します。
2.3.0	新機能 • Linux での Greengrass nucleus の HTTP プロキシ設定サポートを追加します。 バグ修正と機能向上 • セキュリティ上の問題を修正します (CVE-2019-19135)。

バージョン	変更

2.2.0	新機能
	• Linux ARMv8 アーキテクチャにデータコレクションパックをインストールするためのサポートを追加します。
	• Linux ARMv8 の最小要件:
	  • メモリ: 4 GB
	  • CPU: ARM Cortex-A72 または同等の仕様
	
	バグ修正と機能向上
	• ノード検出プロセスにおけるメトリクスのログ記録を改善します。
	• サポートされていないデータ型の処理を改善します。
	• データストリームエラーのログ記録を改善します。

2.1.3	新機能 • Windows Server 2019 以降のサポートを追加。 バグ修正と機能向上 • サポートされていないデバイスにこのコンポーネントをデプロイするときのエラーメッセージが改善。

バージョン	変更

2.1.1	新機能
	• 次のサブスクリプションプロパティの設定のサポートが追加されました。
	  • DataChangeTrigger - データ変更アラートを開始する条件を定義できます。
	  • QueueSize - モニタリング対象アイテムの通知がキューに入れられる特定のメトリクスの OPC-UA サーバー上のキューの深さ。
	  • PublishingIntervalMilliseconds - サブスクリプションの作成時に指定された発行サイクルの間隔 (ミリ秒単位) 。
	  • SnapshotFrequencyMilliseconds - AWS IoT SiteWise エッジが安定したデータストリームを取り込むように、スナップショットの頻度タイムアウト設定を構成できます。
	• このバージョンは BAD 品質データの取り込みをサポートし、以下のデータ品質に基づいてデータをフィルタリングします。
	  • UNCERTAIN 品質データ
	  • BAD 品質データ
	バグ修正と機能向上
	• 顧客メトリクスの改善。
	• 暗号化を有効にしてサーバーに接続するときに問題が発生することがあるセキュリティエンコーディングを修正しました。
	• プロパティグループの更新に失敗する問題を修正しました。

2.0.3	バグ修正と機能向上。
2.0.2	エッジとのアセット優先度同期のバグ修正と機能向上。
2.0.1	当初のバージョン

以下も参照してください。

- 「 ユーザーガイド」の「 AWS IoT SiteWiseとは」。 AWS IoT SiteWise

IoT SiteWise OPC-UA データソースシミュレーター

IoT SiteWise OPC-UA データソースシミュレーターコンポーネント (`aws.iot.SiteWiseEdgeOpcuaDataSourceSimulator`) は、サンプルデータを生成するローカル OPC-UA サーバーを起動します。この OPC-UA サーバーを使用して、 AWS IoT SiteWiseゲートウェイ上の IoT SiteWise OPC-UA コレクターコンポーネントによって読み取られるデータソースをシミュレートします。その後、サンプルデータを使用して、AWS IoT SiteWise 機能を確認できます。AWS IoT SiteWise ゲートウェイについての詳細は、「AWS IoT SiteWise ユーザーガイド」の「エッジで AWS IoT SiteWise を使用する」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 1.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Greengrass コアデバイスは、ローカルホストのポート 4840 を使用できる必要があります。このコンポーネントのローカル OPC-UA サーバーは、このポートで動作します。

依存関係

コンポーネントをデプロイするとき、AWS IoT Greengrass はそれと互換性のあるバージョンの依存関係もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

次の表では、このコンポーネントのすべてのバージョンの依存関係を一覧表示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.3.0 <3.0.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントに設定パラメータはありません。

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/aws.iot.SiteWiseEdgeOpcuaDataSourceSimulator.log
```

Windows

```
C:\greengrass\v2\logs\aws.iot.SiteWiseEdgeOpcuaDataSourceSimulator.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/
  aws.iot.SiteWiseEdgeOpcuaDataSourceSimulator.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs
  \aws.iot.SiteWiseEdgeOpcuaDataSourceSimulator.log -Tail 10 -Wait
  ```

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
1.0.0	当初のバージョン Windows Server 2016 以降のサポートを追加。

以下も参照してください。

- 「AWS IoT SiteWise ユーザーガイド」の「AWS IoT SiteWise とは」。

IoT SiteWise パブリッシャー

IoT SiteWise パブリッシャーコンポーネント (`aws.iot.SiteWiseEdgePublisher`) を使用すると、 AWS IoT SiteWise ゲートウェイはエッジから にデータをエクスポートできます AWS クラウド。

AWS IoT SiteWise ゲートウェイの詳細については、「 AWS IoT SiteWise ユーザーガイド」の「エッジ AWS IoT SiteWise での の使用」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

> **Note**
>
> IoT SiteWise パブリッシャーバージョン 3.1.1 は廃止されました。バージョン 3.1.0 を使用することをお勧めします。

- 3.1.x
- 3.0.x
- 2.4.x
- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Greengrass コアデバイスは、次のいずれかのプラットフォームで実行する必要があります。
  - OS: Ubuntu 18.04 以降

    アーキテクチャ: x86_64 (AMD64) または ARMv8 (Aarch64)
  - OS: Red Hat Enterprise Linux (RHEL) 8

    アーキテクチャ: x86_64 (AMD64) または ARMv8 (Aarch64)
  - OS: Amazon Linux 2

    アーキテクチャ: x86_64 (AMD64) または ARMv8 (Aarch64)

- OS: Debian 11

  アーキテクチャ: x86_64 (AMD64) または ARMv8 (Aarch64)

- OS: Windows Server 2019 以降

  アーキテクチャ: x86_64 (AMD64)

- Greengrass コアデバイスはインターネットに接続する必要があります。
- Greengrass コアデバイスは、`iotsitewise:BatchPutAssetPropertyValue` アクションを実行する権限を付与されている必要があります。詳細については、「コアデバイスが AWS サービスとやり取りすることを許可する」を参照してください。

  Example アクセス許可ポリシー

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iotsitewise:BatchPutAssetPropertyValue",
            "Resource": "*"
        }
    ]
}
```

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
`data.iotsitewise.`*region*`.amazonaws.com`	443	はい	データを に発行します AWS IoT SiteWise。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

次の表に、このコンポーネントのバージョン 2.0.x から 2.2.x までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
Greengrass nucleus	>=2.3.0<3.0.0	ハード
ストリームマネージャー	>=2.0.10<3.0.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントに設定パラメータはありません。

AWS IoT SiteWise コンソールまたは API を使用して IoT IoT SiteWise パブリッシャーコンポーネントを設定できます。詳細については、「AWS IoT SiteWise ユーザーガイド」の「ステップ 3: パブリッシャーの設定 (オプション)」を参照してください。

入力データ

このコンポーネントは、 AWS IoT Greengrass ストリームマネージャーから`PutAssetPropertyValueEntry`メッセージを読み取ります。詳細については、AWS IoT SiteWise API リファレンスの PutAssetPropertyValueEntry を参照してください。

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/aws.iot.SiteWiseEdgePublisher.log
```

Windows

```
C:\greengrass\v2\logs\aws.iot.SiteWiseEdgePublisher.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* を AWS IoT Greengrass ルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.iot.SiteWiseEdgePublisher.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.iot.SiteWiseEdgePublisher.log -Tail 10 -
  Wait
  ```

トラブルシューティングとデバッグ

このコンポーネントには、お客様が問題を特定して解決するのに役立つ新しいイベントログが含まれています。ログファイルはローカルログファイルとは別のもので、次の場所にあります。*/greengrass/v2* または *C:\greengrass\v2* を AWS IoT Greengrass ルートフォルダへのパスに置き換えます。

Linux

```
/greengrass/v2/work/aws.iot.SiteWiseEdgePublisher/logs/
IotSiteWisePublisherEvents.log
```

Windows

```
C:\greengrass\v2\work\aws.iot.SiteWiseEdgePublisher\logs
\IotSiteWisePublisherEvents.log
```

このログには、詳細情報とトラブルシューティングの手順が含まれています。トラブルシューティングに関する情報は診断結果とともに提供され、問題の解決方法の説明と、場合によっては詳細情報へのリンクが含まれます。診断情報には次のものが含まれます。

- 重要度レベル
- タイムスタンプ
- イベント固有の追加情報

Example ログの例

```
accountBeingThrottled:
  Summary: Data upload speed slowed due to quota limits
  Level: WARN
  Timestamp: '2023-06-09T21:30:24.654Z'
  Description: The IoT SiteWise Publisher is limited to the "Rate of data points
 ingested"
    quota for a customers account. See the associated documentation and associated
    metric for the number of requests that were limited for more information. Note
    that this may be temporary and not require any change, although if the issue
 continues
    you may need to request an increase for the mentioned quota.
  FurtherInformation:
  - https://docs.aws.amazon.com/iot-sitewise/latest/userguide/quotas.html
  - https://docs.aws.amazon.com/iot-sitewise/latest/userguide/troubleshooting-
gateway.html#gateway-issue-data-streams
  AssociatedMetrics:
  - Name: TotalErrorCount
    Description: The total number of errors of this type that occurred.
    Value: 327724.0
  AssociatedData:
  - Name: AggregatePropertyAliases
    Description: The aggregated property aliases of the throttled data.
    FileLocation: /greengrass/v2/work/aws.iot.SiteWiseEdgePublisher/./logs/data/
AggregatePropertyAliases_1686346224654.log
```

ライセンス

このコンポーネントは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされています。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
3.1.1	バグ修正と機能向上 • エラーが発生したときに、影響を受けるデータエイリアスを識別する追加のログ記録を追加します。 • 取り込まれたデータの経過時間に対する AWS IoT SiteWise API 制限をローカルで適用するようになりました。 • Amazon S3 の送信先が複数ある場合にパブリッシャーが StreamManager ストリームのチェックポイントを混在させる問題を修正しました。 • パブリッシャーが StreamManager ストリームから読み取る方法に関するパフォーマンスのボトルネックを修正しました。
3.1.0	新機能 • Amazon S3 にデータを Parquet ファイルとして公開するサポートを追加しました。 • AWS IoT SiteWise バッファされた取り込みのサポートを追加しました。
3.0.0	バグ修正と機能向上 • プロキシサポートに関連する問題を修正しました。 新機能 • MQTT ブローカーからのデータ取り込みのサポートを有効にします。
2.4.1	バグ修正と機能向上 • コンポーネントが Java Corretto 11 バージョン11.0.20.8.1 以降で動作できるようにします。コンポーネントバージョン 2.4.0 および 2.3.3 を Java Corretto バージョン 11.0.20.8.1 とともに使用すると、"Could

バージョン	変更
	not find or load main class" エラーメッセージが表示されます。
2.4.0	新機能 • 新しいイベントログを追加して、問題の特定と修正を容易にしています。 バグ修正と機能向上 • パブリッシャーのチェックポイントリカバリを改善しました。
2.3.3	バグ修正と機能向上 • 高スループットへの対応を改善しました。
2.3.2	バグ修正と機能向上 • パブリッシャー設定をダウンロードする際の HTTP プロキシのサポートを修正しました。

2.3.1	新機能
	• Linux ARMv8 アーキテクチャにデータコレクションパックをインストールするためのサポートを追加します。
	• Linux ARMv8 の最小要件:
	  • メモリ: 4 GB
	  • CPU: ARM Cortex-A72 または同等の仕様

2.2.3	バグ修正と機能向上 • 再試行可能な例外リストにない汎用例外の再試行を削除します。
2.2.2	バグ修正と機能向上 • HTTP プロキシサーバー AWS IoT SiteWise 経由で へのデータアップロードサポートを再導入します。

バージョン	変更
2.2.1	**Note** このバージョンは、HTTP プロキシ設定をサポートしていません。バージョン 2.2.2 以降では、この機能のサポートが再導入されています。 新機能 • このコンポーネントにサポートを追加して、データを AWS IoT SiteWiseにアップロードするときに圧縮を切り替えます。

2.2.0	**Note**
	このバージョンは、HTTP プロキシ設定をサポートしていません。バージョン 2.2.2 以降では、この機能のサポートが再導入されています。
	
	新機能
	• データを圧縮してから AWS IoT SiteWise サービスに送信するようにこのコンポーネントを更新。
	  • ほとんどの場合、この変更により、コンポーネントの以前のバージョンと比較して、帯域幅の使用量が 75% 削減されます。
	  • ほとんどの場合、この変更により CPU 使用率が最大 5% 増加します。大量のデータを処理するゲートウェイでは、この変更により CPU 使用率が最大 15% 増加する可能性があります。
	  • この変更は、 AWS IoT SiteWise サービス料金やサービスクォータの使用には影響しません。
	  • Windows Server 2019 以降のサポートを追加。
	バグ修正と機能向上
	• チェックポイントファイルが破損しているときにこのコンポーネントを起動できない問題を修正。

バージョン	変更
2.1.4	バグ修正と機能向上 • Java バージョン 8 との互換性を修正。
2.1.3	⚠ Warning このバージョンは、米国東部 (オハイオ)、カナダ (中部)、および AWS GovCloud （米国東部) リージョンを除き、利用できなくなりました。このコンポーネントバージョンを実行するには Java バージョン 11 以上が必要です。このバージョンの改善は、このコンポーネントのそれ以降のバージョンで利用できます。 バグ修正と機能向上 • サポートされていないデバイスにこのコンポーネントをデプロイするときのエラーメッセージが改善。 • データのアップロードが失敗したときにエラーをログに記録するように更新。
2.1.2	バグ修正と機能向上 • データの有効期限が切れ次第、期限切れのデータエクスポート機能を呼び出すように更新。
2.1.1	バグ修正と機能向上。
2.1.0	新機能 • 最新のデータをクラウドに最初に公開するためのサポートを追加。 • 期限切れのデータをクラウドに公開しないためのサポートを追加。 • 期限切れのデータをローカルに保存するためのサポートを追加。 バグ修正と機能向上 • ディスク I/O とそれに対応するレイテンシーを削減。
2.0.2	バグ修正と機能向上。
2.0.1	当初のバージョン

以下も参照してください。

- 「 ユーザーガイド」の「 AWS IoT SiteWiseとは」。 AWS IoT SiteWise

IoT SiteWise プロセッサ

IoT SiteWise プロセッサコンポーネント (`aws.iot.SiteWiseEdgeProcessor`) を使用すると、AWS IoT SiteWise ゲートウェイはエッジでデータを処理できます。

このコンポーネントを使用すると、 AWS IoT SiteWise ゲートウェイはアセットモデルとアセットを使用してゲートウェイデバイスのデータを処理できます。 AWS IoT SiteWise ゲートウェイの詳細については、「 AWS IoT SiteWise ユーザーガイド」の「エッジ AWS IoT SiteWise での の使用」を参照してください。

トピック

バージョン

このコンポーネントには、次のバージョンがあります。

- 3.2.x
- 3.1.x
- 3.0.x
- 2.2.x
- 2.1.x

- 2.0.x

タイプ

このコンポーネントはジェネリックコンポーネント (`aws.greengrass.generic`) です。Greengrass nucleus は、コンポーネントのライフサイクルスクリプトを実行します。

詳細については、「コンポーネントタイプ」を参照してください。

オペレーティングシステム

このコンポーネントは、次のオペレーティングシステムを実行するコアデバイスにインストールできます。

- Linux
- Windows

要件

このコンポーネントには次の要件があります。

- Greengrass コアデバイスは、次のいずれかのプラットフォームで実行する必要があります。
  - OS: Ubuntu 20.04 または 18.04

    Architecture: x86_64 (AMD64)
  - OS: Red Hat Enterprise Linux (RHEL) 8

    Architecture: x86_64 (AMD64)
  - OS: Amazon Linux 2

    Architecture: x86_64 (AMD64)
  - OS: Windows Server 2019 以降

    アーキテクチャ: x86_64 (AMD64)
- Greengrass コアデバイスは、ポート 443 のインバウンドトラフィックを許可する必要があります。
- Greengrass コアデバイスは、ポート 443 と 8883 のアウトバウンドトラフィックを許可する必要があります。

- 次のポートは、 で使用するために予約されています AWS IoT SiteWise: 80、443、3001、4569、4572、8000、8081、8082、8084、8085、8086、8445、9000、9500、11080、および 50010。トラフィック用の予約ポートを使用すると、接続が切断されることがあります。

  > **Note**
  >
  > ポート 8087 が必要になるのは、このコンポーネントのバージョン 2.0.15 以降のみです。

- Greengrass デバイスロールには、 AWS IoT Greengrass V2 デバイスで AWS IoT SiteWise ゲートウェイを使用できるようにするアクセス許可が必要です。詳細については、「AWS IoT SiteWise ユーザーガイド」の「要件」を参照してください。

エンドポイントおよびポート

このコンポーネントは、基本的な操作に必要なエンドポイントとポートに加えて、次のエンドポイントとポートに対し、アウトバウンドリクエストを実行できる必要があります。詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。

エンドポイント	ポート	必要	説明
`model.iotsitewise.region.amazonaws.com`	443	はい	AWS IoT SiteWise アセットとアセットモデルに関する情報を取得します。
`edge.iotsitewise.region.amazonaws.com`	443	はい	コアデバイスの AWS IoT SiteWise ゲートウェイ設定に関する情報を取得します。

エンドポイント	ポート	必要	説明
ecr.*region*.amazonaws.com	443	はい	Amazon Elastic Container Registry から AWS IoT SiteWise Edge ゲートウェイの Docker イメージをダウンロードします。
iot.*region*.amazonaws.com	443	はい	AWS アカウントのデバイスエンドポイントを取得します。
sts.*region*.amazonaws.com	443	はい	の ID を取得します AWS アカウント。
monitor.iotsitewise. *region*.amazonaws.com	443	いいえ	コアデバイスの AWS IoT SiteWise Monitor ポータルにアクセスする場合に必要です。

依存関係

コンポーネントをデプロイすると、 はその依存関係の互換性のあるバージョン AWS IoT Greengrass もデプロイします。つまり、コンポーネントを正常にデプロイするには、コンポーネントとその依存関係のすべての要件を満たす必要があります。このセクションでは、このコンポーネントのリリースされたバージョンの依存関係と、各依存関係に対するコンポーネントのバージョンを定義するセマンティックバージョン制約をリスト表示しています。コンポーネントの各バージョンの依存関係は、AWS IoT Greengrass コンソールでも確認できます。コンポーネントの詳細ページで [Dependencies] (依存関係) リストを確認します。

次の表に、このコンポーネントのバージョン 2.0.x から 2.1.x までの依存関係を示します。

依存関係	互換性のあるバージョン	依存関係タイプ
トークン交換サービス	>=2.0.3 <3.0.0	ハード
ストリームマネージャー	>=2.0.10 <3.0.0	ハード
Greengrass CLI	>=2.3.0 <3.0.0	ハード

コンポーネントの依存関係の詳細については、「コンポーネント recipe のリファレンス」を参照してください。

構成

このコンポーネントに設定パラメータはありません。

ローカルログファイル

このコンポーネントは次のログファイルを使用します。

Linux

```
/greengrass/v2/logs/aws.iot.SiteWiseEdgeProcessor.log
```

Windows

```
C:\greengrass\v2\logs\aws.iot.SiteWiseEdgeProcessor.log
```

このコンポーネントのログを確認するには

- コアデバイスに次のコマンドを実行して、このコンポーネントのログファイルをリアルタイムに確認します。*/greengrass/v2* または *C:\greengrass\v2* を AWS IoT Greengrass ルートフォルダへのパスに置き換えます。

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.iot.SiteWiseEdgeProcessor.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.iot.SiteWiseEdgeProcessor.log -Tail 10 -
  Wait
  ```

ライセンス

このコンポーネントには、次のサードパーティーソフトウェア/ライセンス品が含まれています。

- Apache-2.0
- MIT
- BSD-2-Clause
- BSD-3-Clause
- CDDL-1.0
- CDDL-1.1
- ISC
- Zlib
- GPL-3.0-with-GCC-exception
- パブリックドメイン
- Python-2.0
- Unicode-DFS-2015
- BSD-1-Clause
- OpenSSL
- EPL-1.0

- EPL-2.0
- GPL-2.0-with-classpath-exception
- MPL-2.0
- CC0-1.0
- JSON

このコンポーネントは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされています。

変更ログ

次の表は、コンポーネントの各バージョンにおける変更を示します。

バージョン	変更
3.2.0	フォーマンスの改善 • API サービスを最適化して、メモリフットプリントを小さくし、インストールに必要なディスク容量を削減します。 • これにより、コンポーネント全体の初期メモリ使用量が 2 GB 削減され (起動時に 7.5 GB のメモリが使用されるようになりましたが、16 GB が引き続き推奨されます)、ダウンロードサイズが 500 MB 削減されます (現在は 1.4 GB のダウンロードが必要)。 新機能 • `GetAssetPropertyValueAggregates` API は、エッジで 15 分の集約ウィンドウをサポートするようになりました。 • ポート 8081 および 8082 は、このコンポーネントを正しく実行するために必要なくなりました。 **Note** などの AWS IoT SiteWise データプレーン APIsのローカルエンドポイント を `get-asset-property-value` から `http://localhost:8081` に変更中です`http://localhost:11080/data` 。などの AWS IoT SiteWise コントロールプレーン APIs のローカルエンドポイントを から `http://lo`

バージョン	変更
	`calhost:11080` に変更中ですhttp://localhost:11080/control 。AWS では`list-asset-models` 、SiteWise エッジゲートウェイ HTTPS エンドポイントを使用することをお勧めします。これらのエンドポイントは変更されていません。 バグ修正と機能向上 • からの同期 AWS IoT SiteWise は、前回の同期が中断された場合に、リソースを有効な状態に移行するようになりました。これにより、強制再起動後に一部のリソースが破損する問題が修正されます。 • 同期中にリソースが変更されると、エッジでリソースが破損する稀な状態を修正しました。この条件が検出されると同期が失敗し、リソースは次の同期で再試行されます。 • APIsから呼び出せる問題を修正しました。現在、ローカルループバックアドレスの外部APIs を呼び出すために使用できるのは HTTPS のみです。 • `ListAssets` API で、エッジに保存されているアセットのアセット階層が表示されるようになりました。 • Windows で Data Processing Pack の再起動、アップグレード、またはダウングレードに失敗する問題を修正しました。 • 顧客が認証情報を使用して MQTT ブローカーに接続できない、Windows OS 用データ処理パックのバグを修正しました。
3.1.3	バグ修正と機能向上 • 一部のリソースが実際に失敗したときに、データ処理パックが同期の成功を誤って報告していた問題を修正しました。 • 同じ親を持たない限り、複数のアセットに同じ名前を付けることができます。

バージョン	変更
3.1.1	バグ修正と機能向上 • タイムゾーンの不一致が原因で SigV4 リクエストが失敗する問題を修正しました。 • 再起動後に属性に依存すると、変換プロパティとメトリクスプロパティの計算が停止する問題を修正しました。 • カスタムストリームマネージャーポート設定のサポートを有効にします。 • エッジに同期されたプロパティの更新が停止することがある問題を修正します。
3.1.0	バグ修正と機能向上 • `ListAssetModels` API が次のトークンを生成できない問題を修正しました。
3.0.0	新機能 • MQTT ブローカーからのデータ取り込みのサポートを有効にします。
2.2.1	バグ修正と機能向上 • コントロールプレーンのデータストレージとクラウドの運用方法の一貫性を高めるために、同期プロセスを調整します。これはアップグレードに若干影響します。 **Note** バージョン 2.2.1 以降で同期されたコントロールプレーンデータは、以前のバージョンと互換性がありません。以前のバージョンにダウングレードするには、新規インストールを完了する必要があります。これはアップグレードには影響せず、以前のバージョンで同期されたデータはバージョン 2.2.1 で動作します。 • AWS 認証情報チェーンに追加の変更を加え、 AWS IoT Greengrass V2 認証情報に優先順位を付けます。

バージョン	変更
2.1.37	バグ修正と機能向上 • dependency-routing-service プロセスを非推奨にし、その機能を property-state-service プロセスに移動して、通信するプロセスからのリソース使用量を減らします。 • API の結果の上限を 20,000 `get-asset-property-value-history` に引き上げて、 が使用する制限に一致させます AWS IoT SiteWise。 • 結果の上限が指定されていない場合、ページ分割された結果で `get-asset-property-value-history` API のトークンが提供されない問題を修正しました。
2.1.35	バグ修正と機能向上 • AWS 認証情報チェーンを変更して、認証情報に優先順位 AWS IoT Greengrass を付けます。 • AWS IoT Thing グループの一部としてデプロイするときのアカウント検出の問題を修正しました。
2.1.34	バグ修正と機能向上 • Linux でマルチスレッドを使用するようにメトリック/トランスフォームの計算を調整しました。Windows では、互換性の観点から、引き続きシングルスレッドの計算を実行します。 • 一部の計算ウィンドウでメトリック計算が欠落する問題を修正しました。
2.1.33	バグ修正と機能向上 • Greengrass コンソールへのエラー状態のレポートに関する問題を修正しました。
2.1.32	バグ修正と機能向上 • カスタマイズされたユーザー名とグループのサポートを追加します。
2.1.31	バグ修正と機能向上 • AWS IoT SiteWiseでモデル化されたデータについて、時間加重平均と時間加重標準偏差を計算するためのサポートを追加しました。

バージョン	変更
2.1.29	バグ修正と機能向上 • エッジ機能でアセットをフィルタリングするためのサポートを追加します。
2.1.28	バグ修正と機能向上 • リソースの同期を最適化して、多数のアセットを からエッジ AWS クラウド に同期できるようにします。
2.1.24	バグ修正と機能向上 • 2 回目の同期時にダッシュボードが非表示になる問題を修正しました。
2.1.23	バグ修正と機能向上 • インターネット接続が遅い場合にインストールが失敗しないように、`aws.iot.SiteWiseEdgeProcessor` インストールプロセスのタイムアウトを追加しました。 • クラウドとエッジ間の同期効率を向上させるためにリソース同期を最適化しました。
2.1.21	**Warning** 2.0.x から 2.1.x にアップグレードすると、ローカルデータが失われます。 新機能 • Windows Server 2019 以降のサポートを追加。 • Linux ベースのオペレーティングシステムの Docker を削除。
2.0.16	このバージョンには、バグ修正と機能向上が含まれています。

バージョン	変更
2.0.15	バグ修正と機能向上 • このコンポーネントがリソース同期 API オペレーションに使用するポートを 8085 から 8087 に変更。その結果、このコンポーネントではポート 8087 を利用可能にする必要があります。このコンポーネントには、ポート 8085 が利用可能であることも依然として必要です。 • AWS OpsHub 認証を更新して、ユーザーが API オペレーションを呼び出そうとしたときではなく、ログイン中に許可されていないユーザーを拒否します。
2.0.14	このバージョンには、バグ修正と機能向上が含まれています。
2.0.13	バグ修正と機能向上 • このコンポーネントが Amazon CloudWatch メトリクスにデータをレポートするときに、どのデータがモデル化されていないかが正しく示されるように問題を修正しました。
2.0.9	バグ修正と機能向上 • コアデバイスで AWS IoT SiteWise リソースを作成および更新する信頼性が向上します。 • コアデバイスにインストールされているコンポーネント、各コンポーネントのバージョン、および各コンポーネントのステータスを監視するために使用できるローカル API オペレーションを追加します。この情報は、コアデバイスの アプリケーション用 の「設定 AWS OpsHub AWS IoT SiteWise 」タブで確認できます。 • このコンポーネントが実行する Docker コンテナのヘルスステータスを追加します。`docker ps` コマンドを実行すると、コンテナのヘルスステータスを表示できます。
2.0.7	バグ修正と機能向上 • コアデバイスで AWS IoT SiteWise Monitor ポータルを表示するためのサポートを修正しました。

バージョン	変更
2.0.6	バグ修正と機能向上 • このコンポーネントがコアデバイスで計算する AWS IoT SiteWise `statetime()` 、`earliest()` 、および `latest()`関数を修正しました。
2.0.5	バグ修正と機能向上 • このコンポーネントが AWS IoT SiteWise `pretrigger()` コアデバイスで計算する変換で 関数のサポートを追加しました。 • このコンポーネントが認証用の LDAP (ライトウェイトディレクトリアクセスプロトコル) 設定を格納するパスを変更します。
2.0.2	当初のバージョン

以下も参照してください。

- 「 ユーザーガイド」の「 AWS IoT SiteWiseとは」。 AWS IoT SiteWise

パブリッシャーがサポートするコンポーネント

パブリッシャーがサポートするコンポーネントは、 のプレビューリリースに含まれAWS IoT Greengrassており、変更される可能性があります。これらのコンポーネントは ではサポートされていませんAWS。各コンポーネントに関する問題については、パブリッシャーに連絡する必要があります。

Greengrass Publisher がサポートするコンポーネントは、サードパーティーコンポーネントベンダーによって開発、提供、および提供されます。サードパーティーコンポーネントベンダーは、 AWS Partner Device Catalog、AWSMediaes、またはコミュニティベンダーのいずれかのベンダーのものです。このカタログ内のコンポーネントは、サードパーティーのコンポーネントベンダーに直接連絡することで購入できます。

Greengrass Publisher がサポートするコンポーネントには以下が含まれます。

トピック

- AIShield.Edge
- AI EdgeLabs センサー
- Greengrass S3 インジェスト

AIShield.Edge

このコンポーネントは開発され、Bosch を搭載した AIShield によってサポートされています。AIShield.Edge. このコンポーネントは、脅威に対応したカスタマイズされた防御をエッジデバイスにシームレスにデプロイするように設計されており、デバイスを AI 攻撃から保護します。

このコンポーネントには、次の利点があります。

- AIShield AI Security による脆弱性分析から 内の強化されたエッジディフェンスにシームレスに移行する AWS
- カスタマイズされた防御を複数のエッジデバイスに簡単にデプロイする
- さまざまなモデルタイプやフレームワークをサポートする多様な AI セットアップに合わせた幅広い保護
- Amazon SageMaker と Greengrass ワークフローへのシームレスな統合により更新を維持する
- データを直接 に中継して、潜在的な脅威に関する洞察をすぐに得る AWS IoT Core
- AWS Marketplace の AIShield AI Security からエッジへの防御デプロイのための統一された AI セキュリティパス

このコンポーネントは、次のプラットフォームで実行する必要があります。

- OS: Linux

このコンポーネントの購入に関心がある場合は、ボッシュソフトウェアとデジタルソリューション: <AIShield.Contact@bosch.com> にお問い合わせください。

AI EdgeLabs センサー

このコンポーネントは AI によって開発され、サポートされています EdgeLabs。AI EdgeLabs センサーは、AI ベースの脅威の検出および防止機能を含むコンテナベースのアプリケーションです。AI センサーは Greengrass コンポーネントにラップされ、他の Greengrass コンポーネントと一緒にコアデバイスにスタンドアロンコンテナとしてデプロイされます。

この現在のコンポーネントは、ネットワーク通信を継続的に検証し、エッジホストまたは IoT ゲートウェイで実行されているソフトウェアの脅威パターンを検索するコンテナベースのエージェントです。このコンポーネントは、eBPF、プロセス帯域幅の動作検証、ホストベースの設定を使用します。このコンポーネントの主な機能は、NDR/IPS および EDR 関数に基づいています。

このコンポーネントには、次の利点があります。

- ネットワーク攻撃とマルウェア (EDR/NDR) に対する AI ベースの脅威検出
- AI ベースの自動インシデント対応 (IPS)
- 外部へのデータ転送を最小限に抑えたホストローカル脅威インテリジェンス
- Docker と Greengrass による軽量デプロイ

このコンポーネントは、次のいずれかのプラットフォームで実行する必要があります。

- OS: Linux

このコンポーネントの購入に関心がある場合は、AI EdgeLabs: <contact@edgelabs.ai> にお問い合わせください。

Greengrass S3 インジェスト

このコンポーネントは Nathan Glover によって開発され、サポートされています。Greengrass S3 Ingestor コンポーネントは、ストリームマネージャーコンポーネント で使用するように設計されています。このコンポーネントは、ストリームマネージャーから JSON メッセージの行区切りストリームを受け取り、GZIP ファイルにバッチ処理します。このコンポーネントを使用すると、Amazon S3 に効率的にデータを取り込み、さらなる処理や保存を行うことができます。このコンポーネントは、 AWS クラウド へのデータのリアルタイム送信をサポートしていません。

このコンポーネントは、次のいずれかのプラットフォームで実行する必要があります。

- OS: Linux
- OS: Windows

このコンポーネントの購入に関心がある場合は、 Nathan Glover: <nathan@glovers.id.au> にお問い合わせください。

コミュニティコンポーネント

Greengrass Software Catalog は、Greengrass コミュニティによって開発された Greengrass コンポーネントのインデックスです。このカタログから、コンポーネントをダウンロード、変更、デプロイして Greengrass アプリケーションを作成できます。カタログは、 のリンクで確認できます https://github.com/aws-greengrass/aws-greengrass-software-catalog。

各コンポーネントには、調査できるパブリック GitHub リポジトリがあります。コミュニティコンポーネントの完全なリストについては GitHub 、 の Greengrass Software Catalog を参照してください。たとえば、このカタログには、次のコンポーネントが含まれています。

- Amazon Kinesis Video Streams

  このコンポーネントは、リアルタイムストリーミングプロトコル (RTSP) を使用するローカルカメラからオーディオストリームとビデオストリームを取り込みます。次に、コンポーネントはオーディオストリームとビデオストリームを Amazon Kinesis Video Streams にアップロードします。
- Bluetooth IoT ゲートウェイ

  このコンポーネントは、Bluetooth Low Energy (LE) デバイスとの通信を可能にする BluePyライブラリを使用して Bluetooth LE クライアントインターフェイスを作成します。
- Certificate Rotator

  このコンポーネントは、AWS IoT Greengrass コアデバイスの証明書と秘密鍵を、お客様のフリート全体で、大規模にローテーションする手段を提供します。
- コンテナ化されたセキュアトンネリング

  このコンポーネントは、特定のホストオペレーティングシステムに依存しない再利用可能な recipe で、すべての依存関係と一致するライブラリを安全にトンネリングするための Docker コンテナを提供します。
- Grafana

  このコンポーネントを使用すると、Grafana サーバーを Greengrass コアデバイスでホストすることができます。Grafana のダッシュボードを使用して、コアデバイス上のデータを視覚化して管理できます。
- GStreamer for Amazon Lookout for Vision

  このコンポーネントには GStreamer プラグインが用意されているため、カスタム GStreamer パイプラインで Lookout for Vision の異常検出を実行できます。

- Home assistant

  このコンポーネントにより、お客様は Home assistant スマートホームデバイスのローカル制御を提供します。エッジとクラウドで AWS サービスとの統合を提供し、Home Assistant を拡張するホームオートメーションソリューションを提供します。

- InfluxDBGrafana ダッシュボード

  このコンポーネントは、InfluxDB と Grafana コンポーネントをセットアップするワンクリックエクスペリエンスを提供します。InfluxDB を Grafana に接続し、リアルタイムで AWS IoT Greengrass テレメトリをレンダリングするローカルの Grafana ダッシュボードのセットアップを自動化します。

- InfluxDB

  このコンポーネントは、Greengrass コアデバイス上に InfluxDB の時系列データベースを提供するものです。このコンポーネントを使用して、IoT センサーからのデータを処理し、リアルタイムでデータを分析し、エッジでの動作を監視できます。

- InfluxDB パブリッシャー

  このコンポーネントは、Nucleus エミッタプラグイン から InfluxDB に 1 つのシステムヘルステレメトリを中継します。このコンポーネントは、カスタムテレメトリを InfluxDB に転送することもできます。

- IoT pubsub フレームワーク

  このフレームワークは、アプリケーションアーキテクチャ、テンプレートコード、デプロイ可能な例を提供します。これは、AWS IoT Greengrass v2 カスタムコンポーネントを使用して、分散イベント駆動型 IoT pubsub アプリケーションのコード品質を向上させるのに役立ちます。詳細については、「AWS IoT Greengrass コンポーネントの作成」を参照してください。

- Jupyter Lab

  このコンポーネントは AWS IoT Greengrassコアデバイスにデプロイ JupyterLab されます。Jupyter 環境は、AWS IoT Greengrass で設定されたプロセスと環境変数のリソースにアクセスできるため、Python で記述されたコンポーネントのテストと開発のプロセスが簡素化されます。

- ローカル Web サーバー

  このコンポーネントを使用すると、Greengrass コアデバイスにローカル Web ユーザーインターフェイスを作成できます。たとえば、デバイスおよびアプリケーションの設定を設定したり、デバイスを監視できるローカル Web ユーザインターフェイスを作成したりすることができます。

- LoRaWaN プロトコルアダプター

  このコンポーネントは、省電力広域ネットワーク (LPWAN) プロトコルである LoRaWaN プロトコルを使用するローカルワイヤレスデバイスからデータを取り込みます。このコンポーネントを使用することで、クラウドと通信することなく、ローカルでデータを分析して処理できます。

- Modbus TCP

  このコンポーネントは、ModbusTCP プロトコルを使用してローカルデバイスからデータを収集し、選択したデータストリームに公開します。

- Node-RED

  このコンポーネントは、NPM を使用して AWS IoT Greengrass コア デバイスに Node-RED をインストールします。このコンポーネントは、明示的にデプロイおよび構成する必要がある Node-RED Auth コンポーネントに依存します。Node-RED CLI for Greengrass を使用して、Node-RED フローを AWS IoT Greengrass デバイスにデプロイできます。

- Node-RED Docker

  このコンポーネントは、公式の Node-RED Docker コンテナを使用して、AWS IoT Greengrass コアデバイスに Node-RED をインストールします。このコンポーネントは、明示的にデプロイおよび構成する必要がある Node-RED Auth コンポーネントに依存します。Node-RED CLI for Greengrass を使用して、Node-RED フローを AWS IoT Greengrass デバイスにデプロイできます。

- Node-RED Auth

  このコンポーネントは、ユーザー名とパスワードを構成して、AWS IoT Greengrass コアデバイスで実行されている Node-RED インスタンスを保護します。

- OpenThread ボーダールーター

  このコンポーネントは OpenThread ボーダールーター Docker コンテナをデプロイします。このコンポーネントは、Thread のボーダールーターを含む Matter デバイスを構成するのに役立ちます。

- OSI Pi ストリーミングデータコネクタ

  このコンポーネントは、OSI Pi Data Archive から の最新のデータアーキテクチャへのストリーミングリアルタイムデータ取り込みを提供しますAWS。AWS IoT PubSub メッセージングで一元管理される OSI Pi アセットフレームワークと統合されます。

- PostgreSQL DB

このコンポーネントは、エッジで PostgreSQL リレーショナルデータベースのサポートを提供します。お客様はこのコンポーネントを使用して、docker コンテナ内のローカル PostgreSQL インスタンスをプロビジョニングおよび管理できます。

- S3 ファイルアップローダー

  このコンポーネントは、新しいファイルのディレクトリを監視し、それらを Amazon Simple Storage Service (Amazon S3) にアップロードして、アップロードが成功したら削除します。

- Secrets Manager クライアント

  このコンポーネントは、recipe ライフサイクルスクリプトで Secrets Manager コンポーネントからシークレットを取得する必要がある他のコンポーネントで使用できる CLI ツールを提供します。

- コンテナへの TES ルーティング

  このコンポーネントは、コンテナーで トークン交換サービス コンポーネントを使用できるように、AWS IoT Greengrass デバイスで nftables または iptables を構成します。

- WebRTC

  このコンポーネントは、AWS IoT Greengrass コア デバイスに接続された RTSP カメラからオーディオおよびビデオ ストリームを取り込みます。次に、コンポーネントはオーディオストリームとビデオストリームを通信に変換するか、 peer-to-peerAmazon Kinesis Video Streams を介して中継します。

機能をリクエストしたり、バグを報告したりするには、そのコンポーネントのリポジトリで GitHub 問題を開きます。 AWSはコミュニティコンポーネントのサポートを提供していません。詳細については、各コンポーネントのリポジトリの CONTRIBUTING.md ファイルを参照してください。

AWS で提供されているコンポーネントの中にもオープンソースのものがあります。詳細については、「オープンソース AWS IoT Greengrass Core ソフトウェア」を参照してください。

AWS IoT Greengrass 開発用ツール

カスタム Greengrass コンポーネントの作成、テスト、構築、パブリッシュ、デプロイには、AWS IoT Greengrass 開発ツールを使用します。

- Greengrass Development Kit CLI

ローカル開発環境の AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) を使用して、Greengrass Software Catalog のテンプレートおよびコミュニティコンポーネントからコンポーネントを作成します。GDK CLI を使用してコンポーネントをビルドし、AWS アカウント のプライベートコンポーネントとしてコンポーネントを AWS IoT Greengrass サービスにパブリッシュします。

- Greengrass コマンドラインインターフェイス

  Greengrass コアデバイスで Greengrass コマンドラインインターフェイス (Greengrass CLI) を使用して、Greengrass コンポーネントのデプロイとデバッグを行います。Greengrass CLI は、コアデバイスにデプロイできるコンポーネントで、ローカルデプロイの作成、インストールされたコンポーネントの詳細の表示、およびログファイルの検索を行うことができます。

- ローカルデバッグコンソール

  Greengrass コアデバイスのローカルデバッグコンソールで、ローカルダッシュボードウェブインターフェイスを使用して Greengrass コンポーネントをデプロイおよびデバッグします。ローカルデバッグコンソールは、コアデバイスにデプロイしてローカルデプロイを作成し、インストールされているコンポーネントの詳細を表示できるコンポーネントです。

AWS IoT Greengrass には、カスタムの Greengrass コンポーネントで使用が可能な、以下の SDK も用意されています。

- プロセス間通信 (IPC) ライブラリが含まれている AWS IoT Device SDK。詳細については、「AWS IoT Device SDK を使用して Greengrass nucleus、その他のコンポーネント、および AWS IoT Core と通信する」を参照してください。
- データストリームを AWS クラウド に転送するために使用できる、Stream Manager SDK。詳細については、「Greengrass コアデバイスでのデータストリームの管理」を参照してください。

トピック

- AWS IoT Greengrass Development Kit Command-Line Interface
- Greengrass コマンドラインインターフェイス
- AWS IoT Greengrass テストフレームワークを使用する

AWS IoT Greengrass Development Kit Command-Line Interface

AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) は、カスタム Greengrass コンポーネントの開発に役立つ機能を提供します。GDK CLI を使用して、カスタムコンポーネントを作成、ビルドおよびパブリッシュできます。GDK CLI でコンポーネントリポジトリを作成する場合、Greengrass Software Catalog (Greengrass ソフトウェアカタログ) のテンプレートまたはコミュニティコンポーネントから開始できます。次に、ファイルを ZIP アーカイブとしてパッケージ化する、Maven または Gradle 構築スクリプトを使用する、またはカスタム構築コマンドを実行する構築システムを選択できます。コンポーネントの作成後、GDK CLI を使用して AWS IoT Greengrass サービスにパブリッシュして、Greengrass コアデバイスにコンポーネントをデプロイするための AWS IoT Greengrass コンソールまたは API を使用できるようにします。

GDK CLI を使用せずに Greengrass コンポーネントを開発する場合は、コンポーネントの新しいバージョンを作成するたびにコンポーネント recipe ファイルでバージョンとアーティファクト URI を更新する必要があります。GDK CLI を使用すると、新しいバージョンのコンポーネントをパブリッシュするたびに、自動的にバージョンとアーティファクト URI を更新できます。

GDK CLI はオープンソースであり、 で使用できます GitHub。GDK CLI は、コンポーネント開発のニーズに合わせてカスタマイズおよび拡張できます。 GitHub リポジトリで問題を開き、リクエストをプルすることをお勧めします。GDK CLI ソースは、 のリンクにありますhttps://github.com/aws-greengrass/aws-greengrass-gdk-cli。

前提条件

Greengrass Development Kit CLI をインストールして使用するには、次のものが必要です。

- AWS アカウント。アカウントをお持ちでない場合は、「AWS アカウント のセットアップ」を参照してください。
- インターネットに接続された Windows、macOS、または Unix のような開発用コンピュータ。
- GDK CLI バージョン 1.1.0 以降の場合、Python 3.6 以降がインストールされた開発用コンピュータ。

  GDK CLI バージョン 1.0.0 の場合、Python 3.8 以降がインストールされた開発用コンピュータ。
- Git が開発コンピュータにインストールされていること。
- 開発コンピュータに AWS Command Line Interface (AWS CLI) がインストールされており、認証情報が設定されていること。詳細については、「AWS Command Line Interface ユーザーガイド」の「AWS CLI のインストール、更新、アンインストール」と「AWS CLI の設定」を参照してください。

> **Note**
>
> Raspberry Pi または別の 32 ビット ARM デバイスを使用する場合は、AWS CLI V1.AWS CLI をインストールします。V2 は 32 ビット ARM デバイスでは利用できません。詳細については、「AWS CLI バージョン 1 のインストール、更新、およびアンインストール」を参照してください。

- GDK CLI を使用して AWS IoT Greengrass サービスにコンポーネントをパブリッシュするには、次の権限が必要です。
  - `s3:CreateBucket`
  - `s3:GetBucketLocation`
  - `s3:PutObject`
  - `greengrass:CreateComponentVersion`
  - `greengrass:ListComponentVersions`
- GDK CLI を使用して、ローカルファイルシステムではなく S3 バケット内にアーティファクトが存在するコンポーネントを構築するには、以下の権限が必要です。
  - `s3:ListBucket`

  この機能は GDK CLI v1.1.0 以降で利用できます。

変更ログ

次の表に、GDK CLI の各バージョンの変更をまとめています。詳細については、「」の「GDK CLI リリース」ページを参照してください GitHub。

バージョン	変更
1.6.2	バグ修正と機能向上 • 相対パスが原因で Windows gradlew.bat が機能しない問題を修正しました。 • ログ記録、テスト、パッケージングの軽微な改善。
1.6.1	バグ修正と機能向上 • CLI 引数解析のセキュリティ修正を追加しました。

バージョン	変更
	• GDK が、最新の Greengrass Testing Framework (GTF) リリース名をデフォルトの GTF バージョンとして取得できるようにします。 • GDK が、最新バージョンに更新する古いバージョンの GTF を使用する顧客を推奨できるようにします。
1.6.0	新機能 • `component build` および `component publish` コマンド中に Greengrass レシピスキーマに対するレシピ検証チェックを追加します。この更新により、デベロッパーはコンポーネント作成プロセスの早い段階でコンポーネントレシピ内の実用的な問題を特定できます。 • `test-e2e init` コマンドでプルダウンできる信頼度テストスイートをテンプレートに追加します。この信頼度テストスイートには、基本コンポーネントテストのニーズに合わせて使用および拡張できる 8 つの汎用テストが含まれています。 バグ修正と機能向上 • `test-e2e` コマンドで使用されるデフォルトの Greengrass Testing Framework (GTF) バージョンをバージョン 1.2.0 に更新します。
1.5.0	バグ修正と機能向上 が のときに`excludes`ビルドオプションで認識されるパターンを更新しますbuild_system zip。このバージョンでは、ワイルドカード文字に基づいてパス名と一致する glob パターンが認識されるようになりました。これにより、除外するディレクトリのカスタム仕様が有効になります。

バージョン	変更
1.4.0	新機能 • 既存の GDK 設定ファイル内のフィールドを変更するインタラクティブプロンプトを起動する新しい `config` コマンドが追加されています。 • レシピサイズが Greengrass の要件 (16000 バイト未満) 以内であることを確認してから続行するように `gdk component build` および `gdk component publish` コマンドが変更されています。 バグ修正と機能向上 • レシピの構文エラーが原因でビルドを完了できない場合に、認識できるように `gdk component build` コマンドの出力にさらに口ギングを追加します。 • Open Test Framework の名前が Greengrass Testing Framework に変更されたため、`otf-options` および `otf-version` の名前を `gtf-options` および `gtf-version` にそれぞれ変更します。
1.3.0	新機能 • Open Test Framework を使用したコンポーネントの end-to-end テストをサポートする新しい `test-e2e` コマンドを追加しました。 • 新しい設定オプション `zip_name` を追加して、zip ビルドシステムで設定可能な zip ファイル名をサポートします。 • GDK 設定ファイル内の `region` プロパティをオプションにします。 バグ修正と機能向上 • `--name` 引数を使用して GDK プロジェクトを初期化する際に、指定されたテンプレートまたはリポジトリが存在しない場合でも、新しいディレクトリが作成される問題を修正します。
1.2.3	バグ修正と機能向上 • 不正なエラー処理によりバケット作成に失敗する問題を修正しました。 • コンポーネント recipe のリスト構造が削除される不具合を修正しました。

バージョン	変更
1.2.2	バグ修正と機能向上 • recipe のキーは大文字と小文字を区別しなくなりました。 • 新しいバケットを作成する前に、バケットが AWS リージョン に存在し、ユーザーがアクセス可能かどうかを判断するチェックを追加します。ユーザーに `GetBucketLocation` アクセス許可が必要です。 • GDK CLI 設定ファイルの `excludes` キーワードに関する問題を修正しました。
1.2.1	バグ修正と機能向上 • `gdk-config.json` ファイル内のリージョン設定エントリで、カナダ (中部) (`ca-central-1`) の AWS リージョン を受け入れるように変更しました。 • `publish` コマンドに対する `--region` GDK CLI 引数の問題を修正しました。

バージョン	変更
1.2.0	新機能 • options エントリを GDK CLI 構成ファイルの build 構成に追加しました。 zip ビルドシステムを使用する場合、特定のファイルを zip アーティファクトから除外するために、options で excludes を使用できるようにしました。 • Gradle Wrapper を使用してコンポーネントをビルドするために、gradlew ビルドシステムを追加しました。 • gradle のビルドオプションに、Kotlin DSL ビルドファイルのサポートを追加しました。 • GDK CLI 設定ファイル内で、publish 設定に options エントリを追加しました。options の下で file_upload_args がサポートされ、ファイルを Amazon S3 にアップロードする際に、追加の引数を使用可能にしました。 バグ修正と機能向上 • ビルドコマンドの実行前に、Gradle ビルドがクリーンアップを実行しなかった問題を修正しました。 • ビルドコマンドが失敗した際、ビルド処理が終了しなかった問題を修正しました。 • gdk component list コマンドの出力形式を改善しました。

バージョン	変更
1.1.0	新機能 • Gradle のbuild system (ビルドシステム) のサポートが追加されました。 • Windows デバイスでの Maven の build system (ビルドシステム) のサポートが追加されました。 • component publish コマンドに `--bucket` 引数が追加されました。この引数を使用して、GDK CLI がコンポーネントのアーティファクトをアップロードするバケットを正確に指定できます。 • component init コマンドに `--name` 引数が追加されました。このオプションを使用して、GDK CLI がコンポーネントを初期化するフォルダを指定できます。 • S3 バケットには存在するが、ローカルコンポーネントビルドフォルダには存在しないコンポーネントアーティファクトのサポートが追加されました。この機能を使用すると、機械学習モデルなどの大きなコンポーネントアーティファクトの帯域幅コストを削減できます。 バグ修正と機能向上 • component publish (コンポーネントのパブリッシュ) コマンドを更新して、コンポーネントをパブリッシュする前に構築されているかどうかをチェックします。コンポーネントが構築されていない場合は、このコマンドで コンポーネントを構築できるようになりました。 • ZIP ファイル名に大文字が含まれていると、Windows デバイスで zip ビルドシステムが構築できない問題を修正しました。 • 3.8 以前のバージョンの Python を実行するデバイスのログメッセージの形式を改善し、デフォルトのログレベルを `INFO` に変更しました。 • Python の最小バージョン要件を Python 3.6 に変更しました。
1.0.0	当初のバージョン

AWS IoT Greengrass Development Kit Command-Line Interface をインストールまたは更新する

AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) は Python 上に構築されているため、開発用コンピュータへのインストールには `pip` を使用できます。

> **Tip**
>
> GDK CLI は、venv などの Python 仮想環境にインストールすることもできます。詳細については、「Python 3 ドキュメント」の「仮想環境とパッケージ」を参照してください。

GDK CLI のインストールまたは更新

1. 次のコマンドを実行して、GitHubリポジトリ から GDK CLI の最新バージョンをインストールします。

   ```
   python3 -m pip install -U git+https://github.com/aws-greengrass/aws-greengrass-gdk-cli.git@v1.6.2
   ```

   > **Note**
   >
   > GDK CLI の特定のバージョンをインストールするには、*VersionTag* をインストールするバージョンタグに置き換えます。GDK CLI のバージョンタグは、GitHubリポジトリで表示できます。
   >
   > ```
   > python3 -m pip install -U git+https://github.com/aws-greengrass/aws-greengrass-gdk-cli.git@versionTag
   > ```

2. GDK CLI が正常にインストールされたことを確認するには、次のコマンドを実行します。

   ```
   gdk --help
   ```

   gdk コマンドが見つからない場合は、コマンドのフォルダを PATH に追加してください。

   - Linux デバイスで、 /home/*MyUser*/.local/binを PATH に追加し、 をユーザーの名前*MyUser*に置き換えます。
   - Windows デバイスでは、PATH *PythonPath*\\Scriptsに を追加し、 をデバイス上の Python フォルダへのパス*PythonPath*に置き換えます。

GDK CLI を使用して、Greengrass コンポーネントを作成、ビルドおよびパブリッシュできるようになりました。GDK CLI の使用方法の詳細については、「AWS IoT Greengrass Development Kit Command-Line Interface コマンド」を参照してください。

AWS IoT Greengrass Development Kit Command-Line Interface コマンド

AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) は、開発コンピュータで Greengrass コンポーネントを作成、ビルド、パブリッシュするために使用できるコマンドラインインターフェイスを提供します。GDK CLI コマンドは次の形式を使用します。

```
gdk <command> <subcommand> [arguments]
```

GDK CLI をインストールする際に、コマンドラインから GDK CLI を実行できるように、インストーラでパスに gdk が追加されます。

すべてのコマンドに対して、次の引数を使用できます。

- GDK CLI コマンドの詳細については、-h または --help を使用します。
- GDK CLI のどのバージョンがインストールされているかを確認するには、-v または --version を使用します。
- GDK CLI のデバッグに使用できる詳細なログを出力するには、-d または --debug を使用します。

このセクションでは、GDK CLI コマンドについて説明し、各コマンドの例を示します。各コマンドの概要には、その引数とその使用法が示されています。オプションの引数は角括弧で囲んで表示しています。

使用できるコマンド

- コンポーネント
- config
- test-e2e

コンポーネント

Greengrass コンポーネントの作成、ビルド、パブリッシュには、AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) の component コマンドを使用します。

サブコマンド

- 初期化
- buid (構築)
- 公開

- list

初期化

コンポーネントテンプレートまたはコミュニティコンポーネントから Greengrass コンポーネントフォルダを初期化します。

GDK CLI は Greengrass Software Catalog からコミュニティコンポーネントを取得し、 AWS IoT Greengrassのコンポーネントテンプレートリポジトリからコンポーネントテンプレート GitHubを取得します。

> **Note**
>
> GDK CLI v1.0.0 を使用している場合は、このコマンドは空のフォルダで実行する必要があります。GDK CLI がテンプレートまたはコミュニティコンポーネントを現在のフォルダにダウンロードします。
> GDK CLI v1.1.0 以降を使用する場合、`--name` 引数を使用して、GDK CLI がテンプレートまたはコミュニティコンポーネントをダウンロードするフォルダを指定することができます。この引数を使用する場合は、存在しないフォルダを指定します。GDK CLI によってフォルダが作成されます。この引数を指定しなかった場合、GDK CLI は現在のフォルダを使用しますが、このフォルダは空である必要があります。
> コンポーネントが Zip ビルドシステムを使用する場合、GDK CLI は、コンポーネントのフォルダ内の特定のファイルを、コンポーネントフォルダと同じ名前の zip ファイルに圧縮します。例えば、コンポーネントフォルダの名前が `HelloWorld` の場合、GDK CLI は `HelloWorld.zip` という名前の zip ファイルを作成します。コンポーネント recipe では、zip アーティファクト名はコンポーネントフォルダの名前と一致する必要があります。Windows デバイスで GDK CLI バージョン 1.0.0 を使用する場合、コンポーネントフォルダと zip ファイル名には小文字のみを含める必要があります。
> zip ビルドシステムを使用するテンプレートまたはコミュニティコンポーネントをテンプレートまたはコンポーネントとは異なる名前のフォルダに初期化する場合は、コンポーネント recipe で zip アーティファクト名を変更する必要があります。ZIP ファイル名がコンポーネントフォルダの名前と一致するように `Artifacts` および `Lifecycle` の定義を更新します。次の例では、`Artifacts` と `Lifecycle` の定義内の zip ファイル名を強調表示しています。
>
> JSON
>
> ```
> {
> ```

```
  ...
  "Manifests": [
    {
      "Platform": {
        "os": "all"
      },
      "Artifacts": [
        {
          "URI": "s3://BUCKET_NAME/COMPONENT_NAME/
COMPONENT_VERSION/HelloWorld.zip",
          "Unarchive": "ZIP"
        }
      ],
      "Lifecycle": {
        "run": "python3 -u {artifacts:decompressedPath}/HelloWorld/main.py
 {configuration:/Message}"
      }
    }
  ]
}
```

YAML

```
---
...
Manifests:
  - Platform:
      os: all
    Artifacts:
      - URI: "s3://BUCKET_NAME/COMPONENT_NAME/
COMPONENT_VERSION/HelloWorld.zip"
        Unarchive: ZIP
    Lifecycle:
      run: "python3 -u {artifacts:decompressedPath}/HelloWorld/main.py
 {configuration:/Message}"
```

概要

```
$ gdk component init
    [--language]
    [--template]
```

```
    [--repository]
    [--name]
```

引数 (コンポーネントテンプレートから初期化)

- `-l`、`--language` - 指定したテンプレートに使用するプログラミング言語。

  `--repository` または `--language` および `--template` を指定する必要があります。
- `-t`、`--template` - ローカルコンポーネントプロジェクトに使用するコンポーネントテンプレート。利用可能なテンプレートを表示するには、list コマンドを使用します。

  `--repository` または `--language` および `--template` を指定する必要があります。
- `-n`、`--name` - (オプション) GDK CLI がコンポーネントを初期化するローカルフォルダの名前。存在しないフォルダを指定します。GDK CLI によってフォルダが作成されます。

  この機能は GDK CLI v1.1.0 以降で利用できます。

引数 (コミュニティコンポーネントから初期化)

- `-r`、`--repository` - ローカルフォルダにチェックアウトするコミュニティコンポーネント。利用可能なコミュニティコンポーネントを表示するには、list コマンドを使用します。

  `--repository` または `--language` および `--template` を指定する必要があります。
- `-n`、`--name` - (オプション) GDK CLI がコンポーネントを初期化するローカルフォルダの名前。存在しないフォルダを指定します。GDK CLI によってフォルダが作成されます。

  この機能は GDK CLI v1.1.0 以降で利用できます。

出力

次の例は、このコマンドを実行して Python Hello World テンプレートからコンポーネントフォルダを初期化したときに生成される出力を示しています。

```
$ gdk component init -l python -t HelloWorld
[2021-11-29 12:51:40] INFO - Initializing the project directory with a python
 component template - 'HelloWorld'.
[2021-11-29 12:51:40] INFO - Fetching the component template 'HelloWorld-python'
 from Greengrass Software Catalog.
```

次の例は、このコマンドを実行してコミュニティコンポーネントからコンポーネントフォルダを初期化したときに生成される出力を示しています。

```
$ gdk component init -r aws-greengrass-labs-database-influxdb
```

```
[2022-01-24 15:44:33] INFO - Initializing the project directory with a component
 from repository catalog - 'aws-greengrass-labs-database-influxdb'.
[2022-01-24 15:44:33] INFO - Fetching the component repository 'aws-greengrass-labs-
database-influxdb' from Greengrass Software Catalog.
```

buid (構築)

AWS IoT Greengrass サービスにパブリッシュできる recipe とアーティファクトにコンポーネントのソースを構築します。GDK CLI は、GDK CLI configuration file (GDK CLI 設定ファイル) で指定した `gdk-config.json` で構築システムを実行します。このコマンドは、`gdk-config.json` ファイルが存在するフォルダと同じフォルダで実行する必要があります。

このコマンドを実行すると、GDK CLI は、コンポーネントフォルダ内の `greengrass-build` フォルダに recipe とアーティファクトを作成します。GDK CLI は、recipe を `greengrass-build/recipes` フォルダに保存し、アーティファクトを `greengrass-build/artifacts/componentName/componentVersion` フォルダに保存します。

GDK CLI v1.1.0 以降を使用すると、コンポーネント recipe で S3 バケットには存在して、ローカルコンポーネントの構築フォルダには存在しないアーティファクトを指定できます。この機能を使用すると、機械学習モデルなどの大きなアーティファクトを有するコンポーネントを開発するときに、帯域幅の使用量を減らすことができます。

コンポーネントの構築後は、以下のいずれかの操作を実行すると Greengrass コアデバイス上でコンポーネントをテストできます。

- AWS IoT Greengrass Core ソフトウェアの実行場所とは異なるデバイスで開発する場合、Greengrass コアデバイスにデプロイするには、コンポーネントをパブリッシュする必要があります。コンポーネントを AWS IoT Greengrass サービスにパブリッシュし、Greengrass コアデバイスにデプロイします。詳細については、「パブリッシュコマンド」および「デプロイの作成」を参照してください。
- AWS IoT Greengrass Core ソフトウェアを実行しているデバイスと同じデバイスで開発する場合、コンポーネントを AWS IoT Greengrass サービスにデプロイするか、ローカルデプロイを作成してコンポーネントをインストールして実行することができます。ローカルデプロイを作成するには、Greengrass CLI を使用します。詳細については、「Greengrass コマンドラインインターフェイス と ローカルデプロイで AWS IoT Greengrass コンポーネントをテストする」を参照してください。ローカルデプロイを作成するときは、`greengrass-build/recipes` を recipe フォルダとして、`greengrass-build/artifacts` をアーティファクトフォルダとして使用します。

概要

```
$ gdk component build
```

引数

なし

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。

```
$ gdk component build
[2021-11-29 13:18:49] INFO - Getting project configuration from gdk-config.json
[2021-11-29 13:18:49] INFO - Found component recipe file 'recipe.yaml' in the
 project directory.
[2021-11-29 13:18:49] INFO - Building the component 'com.example.PythonHelloWorld'
 with the given project configuration.
[2021-11-29 13:18:49] INFO - Using 'zip' build system to build the component.
[2021-11-29 13:18:49] WARNING - This component is identified as using 'zip' build
 system. If this is incorrect, please exit and specify custom build command in the
 'gdk-config.json'.
[2021-11-29 13:18:49] INFO - Zipping source code files of the component.
[2021-11-29 13:18:49] INFO - Copying over the build artifacts to the greengrass
 component artifacts build folder.
[2021-11-29 13:18:49] INFO - Updating artifact URIs in the recipe.
[2021-11-29 13:18:49] INFO - Creating component recipe in 'C:\Users\MyUser\Documents
\greengrass-components\python\HelloWorld\greengrass-build\recipes'.
```

公開

このコンポーネントを AWS IoT Greengrass サービスにパブリッシュします。このコマンドは、構築したアーティファクトを S3 バケットにアップロードし、recipe 内のアーティファクト URI を更新し、recipe からコンポーネントの新しいバージョンを作成します。GDK CLI は S3 バケットと、GDK CLI configuration file (GDK CLI 設定ファイル) の `gdk-config.json` で指定した AWS リージョンを使用します。このコマンドは、`gdk-config.json` ファイルが存在するフォルダと同じフォルダで実行する必要があります。

GDK CLI v1.1.0 以降を使用する場合、`--bucket` 引数を指定して、GDK CLI がコンポーネントのアーティファクトをアップロードする S3 バケットを指定します。この引数を指定しない場合、GDK CLI は名前が *`bucket`*-*`region`*-*`accountId`* である S3 バケットにアップロードします。ここで

は、gdk-config.json で指定する値は *bucket* と *region* であり、*accountId* は AWS アカウント ID です。GDK CLI は、バケットが存在しない場合に作成します。

GDK CLI v1.2.0 以降を使用する場合には、--region パラメータを使用して、GDK CLI 設定ファイル内で指定されている AWS リージョン を上書きできます。--options パラメータを使用すると、追加のオプションを指定することも可能です。使用可能なオプションのリストについては、「Greengrass Development Kit CLI 設定ファイル」を参照してください。

このコマンドを実行すると、GDK CLI は、recipe で指定したバージョンでコンポーネントをパブリッシュします。NEXT_PATCH を指定した場合、GDK CLI は、まだ存在しない次のパッチバージョンを使用します。セマンティックバージョンは、major.minor.patch という番号方式になっています。詳細については、「セマンティックバージョンの仕様」を参照してください。

> **Note**
>
> GDK CLI v1.1.0 以降を使用する場合、このコマンドを実行すると、GDK CLI はコンポーネントが構築されているかどうかをチェックします。コンポーネントが構築されていない場合は、GDK CLI はコンポーネントをパブリッシュする前にコンポーネントを構築します。

概要

```
$ gdk component publish
    [--bucket] [--region] [--options]
```

引数

- -b,--bucket - (オプション) GDK CLI がコンポーネントアーティファクトを公開する先の S3 バケットの名前を指定します。

  この引数を指定しない場合、GDK CLI は名前が *bucket*-*region*-*accountId* である S3 バケットにアップロードします。ここでは、gdk-config.json で指定する値は *bucket* と *region* であり、*accountId* は AWS アカウント ID です。GDK CLI は、バケットが存在しない場合に作成します。

  GDK CLI は、バケットが存在しない場合に作成します。

  この機能は GDK CLI v1.1.0 以降で利用できます。
- -r、--region – (オプション) コンポーネントを作成する際の、送り先となる AWS リージョン の名前を指定します。この引数は、GDK CLI 設定内にあるリージョン名を上書きしま

この機能は GDK CLI v1.2.0 以降で利用できます。

- -o、--options (オプション) コンポーネントを公開するためのオプションのリストを指定します。この引数は、有効な JSON 文字列か、公開オプションが記載されている JSON ファイルへのファイルパスにする必要があります。この引数は、GDK CLI 設定内のオプションよりも優先されます。

  この機能は GDK CLI v1.2.0 以降で利用できます。

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。

```
$ gdk component publish
[2021-11-29 13:45:29] INFO - Getting project configuration from gdk-config.json
[2021-11-29 13:45:29] INFO - Found component recipe file 'recipe.yaml' in the
 project directory.
[2021-11-29 13:45:29] INFO - Found credentials in shared credentials file: ~/.aws/
credentials
[2021-11-29 13:45:30] INFO - Publishing the component 'com.example.PythonHelloWorld'
 with the given project configuration.
[2021-11-29 13:45:30] INFO - No private version of the component
 'com.example.PythonHelloWorld' exist in the account. Using '1.0.0' as the next
 version to create.
[2021-11-29 13:45:30] INFO - Uploading the component built artifacts to s3 bucket.
[2021-11-29 13:45:30] INFO - Uploading component artifacts to S3 bucket: {bucket}.
 If this is your first time using this bucket, add the 's3:GetObject' permission
 to each core device's token exchange role to allow it to download the component
 artifacts. For more information, see https://docs.aws.amazon.com/greengrass/v2/
developerguide/device-service-role.html.
[2021-11-29 13:45:30] INFO - Not creating an artifacts bucket as it already exists.
[2021-11-29 13:45:30] INFO - Updating the component recipe
 com.example.PythonHelloWorld-1.0.0.
[2021-11-29 13:45:30] INFO - Creating a new greengrass component
 com.example.PythonHelloWorld-1.0.0
[2021-11-29 13:45:30] INFO - Created private version '1.0.0' of the component in the
 account.'com.example.PythonHelloWorld'.
```

list

利用可能なコンポーネントテンプレートとコミュニティコンポーネントのリストを取得します。

GDK CLI は Greengrass Software Catalog からコミュニティコンポーネントを取得し、 AWS IoT Greengrassのコンポーネントテンプレートリポジトリからコンポーネントテンプレート GitHubを取得します。

このコマンドの出力を init コマンドにパスすることで、テンプレートとコミュニティコンポーネントからコンポーネントリポジトリを初期化できます。

概要

```
$ gdk component list
    [--template]
    [--repository]
```

引数

- `-t、--template` - (オプション) 利用可能なコンポーネントテンプレートを一覧表示するには、この引数を指定します。このコマンドは、各テンプレートの名前と言語を *name*-*language* の形式で出力します。たとえば、HelloWorld-python の場合、テンプレート名は HelloWorld そして言語は python になります。
- `-r、--repository` - (オプション) 利用可能なコミュニティコンポーネントリポジトリを一覧表示するには、この引数を指定します。

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。

```
$ gdk component list --template
[2021-11-29 12:29:04] INFO - Listing all the available component templates from
 Greengrass Software Catalog.
[2021-11-29 12:29:04] INFO - Found '2' component templates to display.
1. HelloWorld-python
2. HelloWorld-java
```

config

AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) の `config` コマンドを使用して、設定ファイル `gdk-config.json` の GDK の設定を変更します。

サブコマンド

- 更新

更新

インタラクティブプロンプトを起動して、既存の GDK 設定ファイル内のフィールドを変更します。

概要

```
$ gdk config update
    [--component]
```

引数

- `-c、--component` - `gdk-config.json` ファイル内のコンポーネント関連のフィールドを更新します。この引数は唯一のオプションのため必須です。

出力

次の例は、このコマンドを実行してコンポーネントを設定するときに生成される出力を示しています。

```
$ gdk config update --component
Current value of the REQUIRED component_name is (default:
 com.example.PythonHelloWorld):
Current value of the REQUIRED author is (default: author):
Current value of the REQUIRED version is (default: NEXT_PATCH):
Do you want to change the build configurations? (y/n)
Do you want to change the publish configurations? (y/n)
[2023-09-26 10:19:48] INFO - Config file has been updated. Exiting...
```

test-e2e

AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) の `test-e2e` コマンドを使用して、GDK プロジェクトで end-to-end テストモジュールを初期化、構築、実行します。

サブコマンド

- 初期化
- buid (構築)
- run

初期化

Greengrass Testing Framework (GTF) を使用するテストモジュールを使用して、既存の GDK CLI プロジェクトを初期化します。

デフォルトでは、GDK CLI は のAWS IoT Greengrassコンポーネントテンプレートリポジトリから Maven モジュールテンプレート GitHubを取得します。この Maven モジュールには、`aws-greengrass-testing-standalone` JAR ファイルに対する依存関係が含まれています。

このコマンドは、GDK プロジェクト内に `gg-e2e-tests` という新しいディレクトリを作成します。テストモジュールディレクトリが既に存在しており、かつ、空でない場合、コマンドは何もせずに終了します。この `gg-e2e-tests` フォルダには、Maven プロジェクトで構造化された Cucumber 機能とステップ定義が含まれています。

デフォルトでは、このコマンドは GTF の最新リリースバージョンを使用しようとします。

概要

```
$ gdk test-e2e init
    [--gtf-version]
```

引数

- `-ov`、 `--gtf-version`- (オプション) GDK プロジェクトの end-to-end テストモジュールで使用する GTF のバージョン。この値は、リリースの GTF バージョンの 1 つである必要があります。この引数は、GDK CLI 設定内の `gtf_version` よりも優先されます。

出力

次の例は、このコマンドを実行し、テストモジュールを使用して GDK プロジェクトを初期化したときに生成される出力を示しています。

```
$ gdk test-e2e init
[2023-12-06 12:20:28] INFO - Using the GTF version provided in the GDK test config
 1.2.0
[2023-12-06 12:20:28] INFO - Downloading the E2E testing template from GitHub into
 gg-e2e-tests directory...
```

buid (構築)

> **Note**
>
> end-to-end テストモジュールを構築するgdk component build前に、 を実行してコンポーネントを構築する必要があります。

end-to-end テストモジュールを構築します。GDK CLI は、`test-e2e` プロパティの下の GDK CLI 設定ファイル `gdk-config.json` で指定したビルドシステムを使用してテストモジュールを構築します。このコマンドは、`gdk-config.json` ファイルが存在するフォルダと同じフォルダで実行する必要があります。

デフォルトでは、GDK CLI は Maven ビルドシステムを使用してテストモジュールを構築します。`gdk test-e2e build` コマンドを実行するには Maven が必要です。

テスト機能ファイルに補間する `GDK_COMPONENT_NAME` や `GDK_COMPONENT_RECIPE_FILE` などの変数がある場合は、テストモジュールを構築する前に gdk-component-build を実行してコンポーネントを構築する必要があります。

このコマンドを実行すると、GDK CLI は GDK プロジェクト設定からすべての変数を補間し、`gg-e2e-tests` モジュールを構築して最終的なテスト JAR ファイルを生成します。

概要

```
$ gdk test-e2e build
```

引数

なし

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。

```
$ gdk test-e2e build
[2023-07-20 15:36:48] INFO - Updating feature file: file:///path/to//
HelloWorld/greengrass-build/gg-e2e-tests/src/main/resources/greengrass/features/
component.feature
[2023-07-20 15:36:48] INFO - Creating the E2E testing recipe file:///path/to/
HelloWorld/greengrass-build/recipes/e2e_test_recipe.yaml
[2023-07-20 15:36:48] INFO - Building the E2E testing module
```

```
[2023-07-20 15:36:48] INFO - Running the build command 'mvn package'
.........
```

run

GDK 設定ファイル内のテストオプションを使用してテストモジュールを実行します。

> **Note**
>
> テストを実行するgdk test-e2e build前に、 を実行して end-to-end テストモジュールを構築する必要があります。

概要

```
$ gdk test-e2e run
    [--gtf-options]
```

引数

- `-oo`、 `--gtf-options` - (オプション) end-to-end テストを実行するためのオプションのリストを指定します。この引数は、有効な JSON 文字列か、GTF オプションが記載されている JSON ファイルへのファイルパスにする必要があります。設定ファイルで指定されたオプションは、コマンド引数で指定されたオプションとマージされます。両方の場所にオプションが存在する場合、引数内のオプションが設定ファイルのオプションよりも優先されます。

  このコマンドで `tags` オプションが指定されていない場合、GDK はタグに `Sample` を使用します。`ggc-archive` が指定されない場合、GDK は最新バージョンの Greengrass nucleus アーカイブをダウンロードします。

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。

```
$ gdk test-e2e run
[2023-07-20 16:35:53] INFO - Downloading latest nucleus archive from url https://
d2s8p88vqu9w66.cloudfront.net/releases/greengrass-latest.zip
[2023-07-20 16:35:57] INFO - Running test jar with command java -jar /path/to/
greengrass-build/gg-e2e-tests/target/uat-features-1.0.0.jar –ggc-archive=/path/to/
aws-greengrass-gdk-cli/HelloWorld/greengrass-build/greengrass-nucleus-latest.zip –
tags=Sample
```

```
16:35:59.693 [] [] [] [INFO]
 com.aws.greengrass.testing.modules.GreengrassContextModule - Extracting /path/
to/workplace/aws-greengrass-gdk-cli/HelloWorld/greengrass-build/greengrass-
nucleus-latest.zip into /var/folders/7g/ltzcb_3s77nbtmkzfb6brwv40000gr/T/gg-
testing-7718418114158172636/greengrass
16:36:00.534 [gtf-1.1.0-SNAPSHOT] [] [] [INFO]
 com.aws.greengrass.testing.features.LoggerSteps - GTF Version is gtf-1.1.0-SNAPSHOT
.......
```

Greengrass Development Kit CLI 設定ファイル

AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) は、`gdk-config.json` という名前の設定ファイルから読み込んで、コンポーネントを構築およびパブリッシュします。この設定ファイルは、コンポーネントリポジトリのルートに存在する必要があります。GDK CLI init コマンドを使用して、この設定ファイルでコンポーネントリポジトリを初期化することができます。

トピック

- GDK CLI 設定ファイルの形式
- GDK CLI 設定ファイルの例

GDK CLI 設定ファイルの形式

コンポーネントの GDK CLI 設定ファイルを定義する場合、次の情報を JSON 形式で指定します。

`gdk_version`

このコンポーネントとの互換性がある GDK CLI の最小バージョン。この値は、リリースの GDK CLI バージョンの 1 つである必要があります。

`component`

このコンポーネントの設定。

`componentName`

`author`

コンポーネントの作成者またはパブリッシャー。

`version`

コンポーネントのバージョン。次のいずれかを指定します。

- `NEXT_PATCH` - このオプションを選択すると、コンポーネントをパブリッシュするときに GDK CLI がバージョンを設定します。GDK CLI は AWS IoT Greengrass サービスを照会して、コンポーネントの最新のパブリッシュバージョンを特定します。次に、そのバージョンの後の次のパッチバージョンにバージョンを設定します。コンポーネントをまだパブリッシュしていない場合は、GDK CLI はバージョン`1.0.0` を使用します。

  このオプションを選択する場合、AWS IoT Greengrass Core ソフトウェアが動作するローカル開発用コンピュータにコンポーネントをデプロイしテストするために、Greengrass CLI を使用することはできません。ローカルデプロイを有効にするには、代わりにセマンティックバージョンを指定する必要があります。

- **`1.0.0`** などのセマンティックバージョンです。セマンティックバージョンは、major.minor.patch という番号方式になっています。詳細については、「セマンティックバージョンの仕様」を参照してください。

  コンポーネントをデプロイしてテストする Greengrass コアデバイスでコンポーネントを開発する場合は、このオプションを選択します。Greengrass CLI を使用してローカルデプロイを作成する場合は、特定のバージョンでコンポーネントを構築する必要があります。

`build`

このコンポーネントのソースをアーティファクトに構築するために使用する設定。このオブジェクトには、次の情報が含まれます。

`build_system`

使用するビルドシステム。次のオプションから選択します。

- `zip` - コンポーネントのフォルダを ZIP ファイルにパッケージ化し、コンポーネントの唯一のアーティファクトとして定義します。次のタイプのコンポーネントには、このオプションを選択します。
  - Python や など、解釈されたプログラミング言語を使用するコンポーネント JavaScript。
  - 機械学習モデルやその他のリソースなどの、コード以外のファイルをパッケージ化するコンポーネント。

  GDK CLI は、コンポーネントのフォルダをコンポーネントフォルダと同じ名前の zip ファイルに圧縮します。例えば、コンポーネントフォルダの名前が `HelloWorld` の場合、GDK CLI は `HelloWorld.zip` という名前の zip ファイルを作成します。

> **Note**
>
> Windows デバイスで GDK CLI バージョン 1.0.0 を使用する場合、コンポーネントフォルダと zip ファイル名には小文字のみを含める必要があります。

GDK CLI がコンポーネントのフォルダを zip ファイルに圧縮する際、次のファイルはスキップされます。

- `gdk-config.json` ファイル
- recipe ファイル (`recipe.json` または `recipe.yaml`)
- ビルドフォルダ (`greengrass-build` など)

- `maven` - `mvn clean package` コマンドを実行して、コンポーネントのソースをアーティファクト内にビルドします。Java コンポーネントなどの Maven を使用するコンポーネントの場合は、このオプションを選択します。

  Windows デバイスでは、この機能は GDK CLI v1.1.0 以降で利用できます。

- `gradle` - `gradle build` コマンドを実行して、コンポーネントのソースをアーティファクト内にビルドします。Gradle を使用するコンポーネントの場合は、このオプションを選択します。この機能は GDK CLI v1.1.0 以降で利用できます。

  `gradle` ビルドシステムは、ビルドファイルとして Kotlin DSL をサポートしています。この機能は GDK CLI v1.2.0 以降で利用できます。

- `gradlew` - `gradlew` コマンドを実行して、コンポーネントのソースをアーティファクト内にビルドします。Gradle Wrapper を使用するコンポーネントの場合は、このオプションを選択します。

  この機能は GDK CLI v1.2.0 以降で利用できます。

- `custom` - カスタムコマンドを実行して、コンポーネントのソースを recipe とアーティファクトにビルドします。`custom_build_command` パラメータでカスタムコマンドを指定します。

`custom_build_command`

(オプション) カスタムビルドシステムに対して実行するカスタムビルドコマンド。`build_system` の `custom` を指定する場合は、このパラメータを指定する必要があります。

> **Important**
>
> このコマンドは、コンポーネントフォルダ内の次のフォルダに recipe とアーティファクトを作成する必要があります。GDK CLI は、コンポーネントビルドコマンドを実行したときにこれらのフォルダを作成します。
>
> - recipe フォルダ: `greengrass-build/recipes`
> - アーティファクトフォルダ: `greengrass-build/artifacts/componentName/componentVersion`
>
> *ComponentName* をコンポーネント名に置き換えて、*ComponentVersion* を コンポーネントバージョンまたはNEXT_PATCH に置き換えます。

1 つの文字列または文字列のリストを指定できます。各文字列が、コマンド内の単語になります。例えば、C++ コンポーネントのカスタムビルドコマンドを実行するには、**`cmake --build build --config Release`** または **`["cmake", "--build", "build", "--config", "Release"]`** を指定する可能性があります。

カスタムビルドシステムの例を表示するには、「」のaws.greengrass.labs.LocalWebServer community component GitHub「」を参照してください。

`options`

(オプション) コンポーネントのビルドプロセス中に使用される追加の設定オプション。

この機能は GDK CLI v1.2.0 以降で利用できます。

`excludes`

zip ファイルを構築するときにコンポーネントディレクトリから除外するファイルを定義する glob パターンのリスト。`build_system` が `zip` の場合にのみ有効です。

> **Note**
>
> GDK CLI バージョン 1.4.0 以前では、除外リストのエントリに一致するファイルは、コンポーネントのすべてのサブディレクトリから除外されます。GDK CLI バージョン 1.5.0 以降で同じ動作を実現するには、除外リストの既存のエントリ`**/`の前に を追加します。例えば、 `*.txt`では、 の

ディレクトリからのみテキストファイルを除外し、 `**/*.txt` では、すべてのディレクトリとサブディレクトリからテキストファイルを除外します。GDK CLI バージョン 1.5.0 以降では、 `excludes`が GDK 設定ファイルで定義されているときに、コンポーネントの構築中に警告が表示されることがあります。この警告を無効にするには、 環境変数を `GDK_EXCLUDES_WARN_IGNORE` に設定します`true`。

GDK CLI は、常に zip ファイルから以下のファイルを除外します。

- `gdk-config.json` ファイル
- recipe ファイル (`recipe.json` または `recipe.yaml`)
- ビルドフォルダ (`greengrass-build` など)

次のファイルはデフォルトで除外されます。ただし、これらのファイルのうち、どのファイルを除外するかは、`excludes` オプションで制御することができます。

- 接頭辞「test」で始まる任意のフォルダ (`test*`)
- 全ての非表示のファイル
- `node_modules` フォルダ

`excludes` オプションを指定すると、GDK CLI は `excludes` オプションで設定したファイルのみを除外します。`excludes` オプションを指定しない場合、GDK CLI は前述のデフォルトのファイルとフォルダを除外します。

`zip_name`

構築プロセス中に zip アーティファクトを作成する際に使用する zip ファイル名。`build_system` が `zip` の場合にのみ有効です。`build_system` が空の場合、コンポーネント名が zip ファイル名として使用されます。

`publish`

このコンポーネントを AWS IoT Greengrass サービスにパブリッシュするために使用する設定。

GDK CLI v1.1.0 以降を使用する場合、`--bucket` 引数を指定して、GDK CLI がコンポーネントのアーティファクトをアップロードする S3 バケットを指定します。この引数を指定しない場合、GDK CLI は名前が *`bucket`*-*`region`*-*`accountId`* である S3 バケットにアップロードします。ここでは、gdk-config.json で指定する値は *`bucket`* と *`region`*

であり、*accountId* は AWS アカウント ID です。GDK CLI は、バケットが存在しない場合に作成します。

このオブジェクトには、次の情報が含まれます。

`bucket`

コンポーネントのアーティファクトをホストするために使用する S3 バケット名。

`region`

GDK CLI がこのコンポーネントをパブリッシュする場所となる AWS リージョン。

GDK CLI v1.3.0 以降を使用している場合、このプロパティはオプションです。

`options`

(オプション) コンポーネントのバージョン作成時に使用される追加の設定オプション。

この機能は GDK CLI v1.2.0 以降で利用できます。

`file_upload_args`

ファイルをバケットにアップロードする際に Amazon S3 に送信される、引数 (メタデータや暗号化メカニズムなど) を含む JSON 構造。使用できる引数のリストについては、Boto3 キュメントの `S3Transfer` クラスを参照してください。

`test-e2e`

(オプション) コンポーネントの end-to-end テスト中に使用する構成。この機能は GDK CLI v1.3.0 以降で利用できます。

`build`

`build_system` – 使用するビルドシステム。デフォルトのオプションは `maven` です。次のオプションから選択します。

- `maven` – `mvn package` コマンドを実行してテストモジュールを構築します。Maven を使用するテストモジュールを構築するには、このオプションを選択します。
- `gradle` – `gradle build` コマンドを実行してテストモジュールを構築します。Gradle を使用するテストモジュールについては、このオプションを選択します。

`gtf_version`

(オプション) GTF で GDK プロジェクトを初期化するときに end-to-end テストモジュールの依存関係として使用する Greengrass Testing Framework (GTF) のバージョン。この値は、リ

リースの GTF バージョンの 1 つである必要があります。デフォルトは GTF バージョン 1.1.0 です。

`gtf_options`

(オプション) コンポーネントの end-to-end テスト中に使用される追加の設定オプション。

次のリストには、GTF バージョン 1.1.0 で使用できるオプションが含まれています。

- `additional-plugins` – (オプション) 追加の Cucumber プラグイン
- `aws-region` – AWS のサービスの特定のリージョンレベルのエンドポイントをターゲットにします。デフォルトは、AWS SDK が検出するものです。
- `credentials-path` – オプションの AWS プロファイル認証情報のパス。デフォルトは、ホスト環境で検出された認証情報です。
- `credentials-path-rotation` – AWS 認証情報のオプションのローテーション期間。デフォルトで 15 分または `PT15M` に設定されます。
- `csr-path` – デバイス証明書の生成に使用される CSR のパス。
- `device-mode` – テスト対象のターゲットデバイス。デフォルトはローカルデバイスです。
- `env-stage` – Greengrass のデプロイ環境をターゲットに設定します。デフォルトは本番です。
- `existing-device-cert-arn` – Greengrass のデバイス証明書として使用する既存の証明書の ARN。
- `feature-path` – 追加の機能ファイルを含むファイルまたはディレクトリ。デフォルトでは、追加の機能ファイルは使用されません。
- `gg-cli-version` – Greengrass CLI のバージョンをオーバーライドします。デフォルトは `ggc.version` にある値です。
- `gg-component-bucket` – Greengrass コンポーネントを格納する既存の Amazon S3 バケットの名前。
- `gg-component-overrides` – Greengrass コンポーネントのオーバーライドのリスト。
- `gg-persist` – テスト実行後に保持されるテスト要素のリスト。デフォルトでは、何も保持しないよう設定されています。許容される値は、`aws.resources`、`installed.software`、および `generated.files` です。
- `gg-runtime` – テストがテストリソースとどのようにインタラクションするかに影響する値のリスト。これらの値は `gg.persist` パラメータよりも優先されます。デフォルトが空の場合、インストールされている Greengrass ランタイムを含む、すべての

テストリソースがテストケースによって管理されると想定されます。許容される値は、`aws.resources`、`installed.software`、および `generated.files` です。

- `ggc-archive` – アーカイブされた Greengrass nucleus コンポーネントへのパス。
- `ggc-install-root` – Greengrass nucleus コンポーネントをインストールするディレクトリ。デフォルトは test.temp.path とテスト実行フォルダです。
- `ggc-log-level` – テスト実行の Greengrass nucleus ログレベルを設定します。デフォルトは「INFO」です。
- `ggc-tes-rolename` – AWS のサービスにアクセスするために AWS IoT Greengrass Core が引き受ける IAM ロール。指定された名前のロールが存在しない場合は、ロールとデフォルトのアクセスポリシーが作成されます。
- `ggc-trusted-plugins` – Greengrass に追加する必要がある、信頼されたプラグインのパス (ホスト上) のカンマ区切りリスト。DUT 自体のパスを指定するには、パスの前に「dut:」というプレフィックスを付けます。
- `ggc-user-name` – Greengrass nucleus の user:group posixUser の値。デフォルトは、ログインしている現在のユーザー名です。
- `ggc-version` – 実行中の Greengrass nucleus コンポーネントのバージョンをオーバーライドします。デフォルトは ggc.archive にある値です。
- `log-level` – テスト実行のログレベル。デフォルトは「INFO」です。
- `parallel-config` – JSON 文字列としてのバッチインデックスとバッチ数のセット。バッチインデックスのデフォルト値は 0、バッチ数は 1 です。
- `proxy-url` – この URL を介してトラフィックをルーティングするようにすべてのテストを設定します。
- `tags` – 機能タグのみを実行します。「&」を使用して組み合わせることができます
- `test-id-prefix` – AWS リソース名とタグを含むすべてのテスト固有のリソースに適用される共通のプレフィックス。デフォルトは「gg」プレフィックスです。
- `test-log-path` – テスト実行全体の結果を含むディレクトリ。デフォルトは「testResults」です。
- `test-results-json` – 結果として得られる Cucumber JSON レポートがディスクに書き込まれた状態で生成されるかどうかを決定するフラグ。デフォルトは true です。
- `test-results-log` – コンソール出力がディスクに書き込まれた状態で生成されるかどうかを決定するフラグ。デフォルトは false です。
- `test-results-xml` – 結果として得られる JUnit XML レポートがディスクに書き込まれた状態で生成されるかどうかを決定するフラグ。デフォルトは true です。

- `test-temp-path` – ローカルテストアーティファクトを生成するディレクトリ。デフォルトは、gg-testing というプレフィックスが付いたランダムな一時ディレクトリです。
- `timeout-multiplier` – すべてのテストタイムアウトに提供される乗数。デフォルトは 1.0 です。

GDK CLI 設定ファイルの例

次の GDK CLI 設定ファイルの例を参照し、Greengrass コンポーネント環境の設定に役立ててください。

Hello World (Python)

次の GDK CLI 設定ファイルは、Python スクリプトを実行する Hello World コンポーネントをサポートしています。この設定ファイルでは、`zip` ビルドシステムを使用して、GDK CLI がアーティファクトとしてアップロードする ZIP ファイルにコンポーネントの Python スクリプトをパッケージ化しています。

```
{
  "component": {
    "com.example.PythonHelloWorld": {
      "author": "Amazon",
      "version": "NEXT_PATCH",
      "build": {
        "build_system" : "zip",
        "options": {
           "excludes": [".*"]
        }
      },
      "publish": {
        "bucket": "greengrass-component-artifacts",
        "region": "us-west-2",
        "options": {
           "file_upload_args": {
              "Metadata": {
                 "some-key": "some-value"
              }
           }
        }
      }
    },
  "test-e2e":{
```

```
    "build":{
        "build_system": "maven"
    },
    "gtf_version": "1.1.0",
    "gtf_options": {
         "tags": "Sample"
     }
  },
  "gdk_version": "1.6.1"
  }
}
```

Hello World (Java)

次の GDK CLI 設定ファイルは、Java アプリケーションを実行する Hello World コンポーネントをサポートしています。この設定ファイルでは、maven ビルドシステムを使用して、GDK CLI がアーティファクトとしてアップロードする JAAR ファイルにコンポーネントの Java ソースコードをパッケージ化しています。

```
{
  "component": {
    "com.example.JavaHelloWorld": {
      "author": "Amazon",
      "version": "NEXT_PATCH",
      "build": {
        "build_system" : "maven"
      },
      "publish": {
        "bucket": "greengrass-component-artifacts",
        "region": "us-west-2",
        "options": {
           "file_upload_args": {
              "Metadata": {
                 "some-key": "some-value"
              }
           }
        }
      }
  },
  "test-e2e":{
    "build":{
        "build_system": "maven"
    },
```

```
    "gtf_version": "1.1.0",
    "gtf_options": {
         "tags": "Sample"
     }
  },
  "gdk_version": "1.6.1"
  }
}
```

コミュニティコンポーネント

Greengrass ソフトウェアカタログの複数のコミュニティコンポーネントで GDK CLI を使用しています。GDK CLI 設定ファイルは、これらのコンポーネントのリポジトリで確認できます。

コミュニティコンポーネントの GDK CLI 設定ファイルを表示するには

1. 次のコマンドを実行して、GDK CLI を使用するコミュニティコンポーネントをリスト表示します。

   ```
   gdk component list --repository
   ```

   レスポンスには、GDK CLI を使用する各コミュニティコンポーネントの GitHub リポジトリの名前が一覧表示されます。各リポジトリは awslabs 組織にあります。

   ```
   [2022-02-22 17:27:31] INFO - Listing all the available component repositories from
    Greengrass Software Catalog.
   [2022-02-22 17:27:31] INFO - Found '6' component repositories to display.
   1. aws-greengrass-labs-database-influxdb
   2. aws-greengrass-labs-telemetry-influxdbpublisher
   3. aws-greengrass-labs-dashboard-grafana
   4. aws-greengrass-labs-dashboard-influxdb-grafana
   5. aws-greengrass-labs-local-web-server
   6. aws-greengrass-labs-lookoutvision-gstreamer
   ```

2. 次の URL でコミュニティコンポーネントの GitHub リポジトリを開きます。を、前のステップのコミュニティコンポーネントの名前*community-component-name*に置き換えます。

   ```
   https://github.com/awslabs/community-component-name
   ```

Greengrass コマンドラインインターフェイス

Greengrass コマンドラインインターフェイス (CLI) を使用するとデバイスで AWS IoT Greengrass Core とやり取りしてコンポーネントをローカルに開発し、問題をデバッグできます。たとえば、Greengrass CLI を使用してローカルデプロイを作成し、コアデバイスでコンポーネントを再起動できます。

Greengrass CLI コンポーネント (`aws.greengrass.Cli`) をデプロイして Greengrass CLI をコアデバイスにインストールします。

> **Important**
>
> このコンポーネントは、本番環境ではなく、開発環境でのみで使用することをお勧めします。このコンポーネントは、通常、本番環境では必要とされない情報や操作へのアクセスを提供します。このコンポーネントを必要なコアデバイスにのみデプロイして、最小特権の原則に従います。

トピック

- Greengrass CLI のインストール
- Greengrass CLI コマンド

Greengrass CLI のインストール

Greengrass CLI は、次のいずれかの方法でインストールできます。

- デバイスで AWS IoT Greengrass Core ソフトウェアを初めてセットアップするときは、 `--deploy-dev-tools`引数を使用します。また、この引数を適用するには `--provision true` も指定する必要があります。
- デバイスに Greengrass CLI コンポーネント (`aws.greengrass.Cli`) をデプロイします。

このセクションでは、Greengrass CLI コンポーネントをデプロイする手順について説明します。初期セットアップ時の Greengrass CLI のインストールについては、「チュートリアル: AWS IoT Greengrass V2 の開始方法」を参照してください。

前提条件

Greengrass CLI コンポーネントをデプロイするには、以下の要件を満たす必要があります。

- AWS IoT Greengrass コアデバイスにインストールおよび設定されたコアソフトウェア。詳細については、「チュートリアル: AWS IoT Greengrass V2 の開始方法」を参照してください。
- を使用して Greengrass CLI を AWS CLI デプロイするには、 をインストールして設定しておく必要があります AWS CLI。詳細については、「AWS CLI ユーザーガイド」の「AWS Command Line Interface の設定。」を参照してください。
- AWS IoT Greengrass Core ソフトウェアを操作するには、Greengrass CLI を使用する権限が必要です。Greengrass CLI を使用するには、次のいずれかを実行します。
  - AWS IoT Greengrass Core ソフトウェアを実行するシステムユーザーを使用します。
  - root 権限または管理者権限を持つユーザーを使用する。Linux コアデバイスでは、`sudo` を使用して root 権限を取得できます。
  - 設定をデプロイする際に、`AuthorizedPosixGroups` または `AuthorizedWindowsGroups` 設定パラメータで指定したグループのシステムユーザーを使用する。詳細については、「Greengrass CLI コンポーネント設定」を参照してください。

Greengrass CLI コンポーネントのデプロイ

以下の手順を完了して、コアデバイスに Greengrass CLI コンポーネントをデプロイします。

Greengrass CLI コンポーネントをデプロイするには (コンソール)

1. AWS IoT Greengrass コンソール にサインインします。
2. ナビゲーションメニューで、[Components] (コンポーネント) を選択します。
3. [Components] (コンポーネント) ページの [Public components] (公開コンポーネント) タブで、[`aws.greengrass.Cli`] を選択します。
4. [aws.greengrass.Cli] ページで、[Deploy] (デプロイ) を選択します。
5. [Add to deployment] (デプロイに追加) から [Create new deployment] (デプロイを新規作成) を選択します。
6. [Specify target] (ターゲットを指定) ページの [Deployment targets] (デプロイターゲット) の [Target Name] (ターゲット名) リストで、デプロイ先にする Greengrass グループを選択し、[Next] (次へ) を選択します。
7. [Select components] (コンポーネントを選択) ページで、aws.greengrass.Cli コンポーネントが選択されていることを確認して、[Next] (次へ) を選択します。
8. [Configure components] (コンポーネント設定) ページで、デフォルト設定のままにして、[Next] (次へ) を選択します。

9. [Configure advanced setting] (高度な設定を設定) ページで、デフォルト構成設定のままにして [Next] (次) を選択します。
10. [Review] (確認) ページで [Deploy] (デプロイ) をクリックします。

Greengrass CLI コンポーネントをデプロイするには (AWS CLI)

1. デバイスで `deployment.json` ファイルを作成して、Greengrass CLI コンポーネントのデプロイ設定を定義します。このファイルは次のようになります:

```
{
  "targetArn":"targetArn",
  "components": {
    "aws.greengrass.Cli": {
      "componentVersion": "2.12.2",
      "configurationUpdate": {
        "merge": "{\"AuthorizedPosixGroups\":\"<group1>,<group2>,...,<groupN>\",
\"AuthorizedWindowsGroups\":\"<group1>,<group2>,...,<groupN>\"}"
      }
    }
  }
}
```

   - [target] フィールドで *targetArn* をデプロイメントの対象となるモノまたはモノのグループの Amazon リソースネーム (ARN) に置き換えます。形式は以下のとおりです:
     - モノ: `arn:aws:iot:`*region*`:`*account-id*`:thing/`*thingName*
     - モノのグループ: `arn:aws:iot:`*region*`:`*account-id*`:thinggroup/`*thingGroupName*
   - `aws.greengrass.Cli` コンポーネントオブジェクトで、以下のように値を指定します。

     `version`

     Greengrass CLI コンポーネントのバージョン。

     `configurationUpdate.AuthorizedPosixGroups`

     (オプション) システムグループのカンマ区切りリストを含む文字列。これらのシステムグループが Greengrass CLI を使用して AWS IoT Greengrass Core ソフトウェアとやり取りすることを許可します。グループ名またはグループ ID を指定できます。例:`group1,1002,group3` は 3 つのシステムグループ (`group1`、`1002` および `group3`) に Greengrass CLI の使用を認可します。

承認するグループを指定しない場合は、ルートユーザー (sudo) または AWS IoT Greengrass Core ソフトウェアを実行するシステムユーザーとして Greengrass CLI を使用できます。

`configurationUpdate.AuthorizedWindowsGroups`

(オプション) システムグループのカンマ区切りリストを含む文字列。これらのシステムグループが Greengrass CLI を使用して AWS IoT Greengrass Core ソフトウェアとやり取りすることを許可します。グループ名またはグループ ID を指定できます。例:`group1,1002,group3` は 3 つのシステムグループ (`group1`、`1002` および `group3`) に Greengrass CLI の使用を認可します。

承認するグループを指定しない場合、Greengrass CLI を管理者または AWS IoT Greengrass Core ソフトウェアを実行するシステムユーザーとして使用できます。

2. 次のコマンドを実行して、デバイスに Greengrass CLI コンポーネントをデプロイします。

```
$ aws greengrassv2 create-deployment --cli-input-json file://path/
to/deployment.json
```

インストール中、コンポーネントはデバイスの */greengrass/v2*/bin フォルダの `greengrass-cli` にシンボリックリンクを追加し、Greengrass CLI の実行はこのパスから行います。Greengrass CLI を絶対パスなしで実行するには、PATH 変数に */greengrass/v2*/bin フォルダを 追加します。Greengrass CLI のインストールを確認するには、以下のコマンドを実行します。

Linux or Unix

```
/greengrass/v2/bin/greengrass-cli help
```

Windows

```
C:\greengrass\v2\bin\greengrass-cli help
```

以下の出力が表示されます。

```
Usage: greengrass-cli [-hV] [--ggcRootPath=<ggcRootPath>] [COMMAND]
Greengrass command line interface
```

```
      --ggcRootPath=<ggcRootPath>
                  The AWS IoT Greengrass V2 root directory.
  -h, --help      Show this help message and exit.
  -V, --version   Print version information and exit.
Commands:
  help                Show help information for a command.
  component           Retrieve component information and stop or restart
                        components.
  deployment          Create local deployments and retrieve deployment status.
  logs                Analyze Greengrass logs.
  get-debug-password  Generate a password for use with the HTTP debug view
                        component.
```

greengrass-cli が見つからない場合は、デプロイで Greengrass CLI のインストールに失敗した可能性があります。詳細については、「トラブルシューティング AWS IoT Greengrass V2」を参照してください。

Greengrass CLI コマンド

Greengrass CLI は、AWS IoT Greengrass コアデバイスとの対話を可能にするコマンドラインインターフェイスを提供します。Greengrass CLI コマンドは次の形式を使用します。

```
$ greengrass-cli <command> <subcommand> [arguments]
```

デフォルトでは、*/greengrass/v2*/bin/ フォルダ内の greengrass-cli 実行可能ファイルは、*/greengrass/v2* フォルダ内で動作している AWS IoT Greengrass Core ソフトウェアのバージョンと対話します。この場所にない実行ファイルを呼び出す場合、または別の場所にある AWS IoT Greengrass Core ソフトウェアとやり取りする場合は、以下のいずれかの方法を使用して、やり取りする AWS IoT Greengrass Core ソフトウェアのルートパスを明示的に指定する必要があります。

- GGC_ROOT_PATH 環境変数を */greengrass/v2* に設定します。
- 次の例のように、コマンドに --ggcRootPath */greengrass/v2* 引数を追加します。

  ```
  greengrass-cli --ggcRootPath /greengrass/v2 <command> <subcommand> [arguments]
  ```

すべてのコマンドに対して、次の引数を使用できます。

- 特定の Greengrass CLI コマンドに関する情報には、--help を使用します。

- Greengrass CLI のバージョンに関する情報には、`--version` を使用します。

このセクションでは、Greengrass CLI コマンドについて説明し、各コマンドの例を示します。各コマンドの概要には、その引数とその使用法が示されています。オプションの引数は角括弧で囲んで表示しています。

使用できるコマンド

- コンポーネント
- デプロイメント
- ログ
- get-debug-password

コンポーネント

`component` コマンドを使用することで、コアデバイス上のローカルコンポーネントとやり取りすることができます。

サブコマンド

- 詳細
- list
- restart
- stop

詳細

1 つのコンポーネントのバージョン、ステータス、および設定を取得します。

概要

```
greengrass-cli component details --name <component-name>
```

引数

`--name`、`-n`。コンポーネントの名前。

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。

```
$ sudo greengrass-cli component details --name MyComponent

Component Name: MyComponent
Version: 1.0.0
State: RUNNING
Configuration: null
```

list

デバイスにインストールされている各コンポーネントの名前、バージョン、ステータス、および設定を取得します。

概要

```
greengrass-cli component list
```

引数

なし

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。

```
$ sudo greengrass-cli component list

Components currently running in Greengrass:
Component Name: FleetStatusService
Version: 0.0.0
State: RUNNING
Configuration: {"periodicUpdateIntervalSec":86400.0}
Component Name: UpdateSystemPolicyService
Version: 0.0.0
State: RUNNING
Configuration: null
Component Name: aws.greengrass.Nucleus
Version: 2.0.0
State: FINISHED
Configuration: {"awsRegion":"region","runWithDefault":
{"posixUser":"ggc_user:ggc_group"},"telemetry":{}}
Component Name: DeploymentService
Version: 0.0.0
```

```
State: RUNNING
Configuration: null
Component Name: TelemetryAgent
Version: 0.0.0
State: RUNNING
Configuration: null
Component Name: aws.greengrass.Cli
Version: 2.0.0
State: RUNNING
Configuration: {"AuthorizedPosixGroups":"ggc_user"}
```

restart

コンポーネントを再起動します。

概要

```
greengrass-cli component restart --names <component-name>,...
```

引数

--names、-n。コンポーネントの名前。少なくとも 1 つのコンポーネント名が必要です。各名前をカンマで区切ることで、追加のコンポーネント名を指定できます。

出力

なし

stop

実行中のコンポーネントを停止します。

概要

```
greengrass-cli component stop --names <component-name>,...
```

引数

--names、-n。コンポーネントの名前。少なくとも 1 つのコンポーネント名が必要です。必要に応じて、各名前をカンマで区切ることで、追加のコンポーネント名を指定できます。

出力

なし

デプロイメント

deployment コマンドを使用することで、コアデバイス上のローカルコンポーネントとやり取りすることができます。

ローカルデプロイの進行状況を監視するには、status サブコマンドを使用します。コンソールを使用してローカルデプロイの進行状況を監視することはできません。

サブコマンド

- 作成
- キャンセル
- list
- status

作成

指定されたコンポーネントレシピ、アーティファクト、ランタイム引数を使用して、ローカルデプロイを作成または更新します。

概要

```
greengrass-cli deployment create
    --recipeDir path/to/component/recipe
    [--artifactDir path/to/artifact/folder ]
    [--update-config {component-configuration}]
    [--groupId <thing-group>]
    [--merge "<component-name>=<component-version>"]...
    [--runWith "<component-name>:posixUser=<user-name>[:<group-name>]"]...
    [--systemLimits "{component-system-resource-limits}]"]...
    [--remove <component-name>,...]
    [--failure-handling-policy <policy name[ROLLBACK, DO_NOTHING]>]
```

引数

- --recipeDir、-r。コンポーネントレシピファイルが格納されているフォルダへのフルパス。

- `--artifactDir`、`-a`。デプロイに含めるアーティファクトファイルが含まれるフォルダへのフルパス。アーティファクトフォルダには、以下のディレクトリ構造が含まれている必要があります。

  ```
  /path/to/artifact/folder/<component-name>/<component-version>/<artifacts>
  ```

- `--update-config`、`-c`。デプロイの設定引数。JSON 文字列または JSON ファイルとして提供されます。JAON 文字列の形式は次のようになります。

  ```
  { \
    "componentName": { \
      "MERGE": {"config-key": "config-value"}, \
      "RESET": ["path/to/reset/"] \
    } \
  }
  ```

  MERGE および RESET は大文字と小文字を区別するため、大文字にする必要があります。

- `--groupId`、`-g`。デプロイのターゲットとなるモノグループ。
- `--merge`、`-m`。追加または更新するターゲットコンポーネントの名前とバージョン。コンポーネント情報は、*<component>*=*<version>* の形式で提供する必要があります。指定する追加コンポーネントごとに個別の引数を使用します。必要に応じて、`--runWith` 引数を指定して、コンポーネントを実行するための `posixUser`、`posixGroup`、および `windowsUser` 情報を提供します。
- `--runWith`。汎用コンポーネントまたは Lambda コンポーネントを実行するための `posixUser`、`posixGroup`、および `windowsUser` 情報。*<component>*:{`posixUser`|`windowsUser`}=*<user>*[:*<=posixGroup>*] 形式でこの情報を提供する必要があります。たとえば、**HelloWorld:posixUser=ggc_user:ggc_group** や **HelloWorld:windowsUser=ggc_user** と指定することができます。指定する追加オプションごとに個別の引数を使用します。

  詳細については、「コンポーネントを実行するユーザーを設定する」を参照してください。

- `--systemLimits`。コアデバイス上の汎用および非コンテナ型 Lambda コンポーネントのプロセスに適用されるシステムリソースの制限。各コンポーネントのプロセスが使用できる CPU および RAM の最大使用数を設定できます。シリアル化された JSON オブジェクトまたは JSON ファイルへのファイルパスを指定します。JSON オブジェクトは次の形式である必要があります。

```
{  \
  "componentName": { \
    "cpus": cpuTimeLimit, \
    "memory": memoryLimitInKb \
  } \
}
```

各コンポーネントに対して、次のシステムリソース制限を設定できます。

- `cpus` – このコンポーネントのプロセスがコアデバイスで使用できる CPU 時間の最大量。コアデバイスの合計 CPU 時間は、デバイスの CPU コア数と同じです。例えば、4 つの CPU コアを持つコアデバイスの場合は、この値を `2` に設定することで、このコンポーネントのプロセスを各 CPU コアの 50% の使用率に制限することができます。CPU コアが 1 つのデバイスの場合は、この値を `0.25` に設定することで、このコンポーネントのプロセスを CPU の 25% の使用率に制限することができます。この値を CPU コア数よりも大きい数値に設定した場合、 AWS IoT Greengrass Core ソフトウェアはコンポーネントの CPU 使用率を制限しません。
- `memory` – このコンポーネントのプロセスがコアデバイスで使用できる RAM の最大量 (キロバイト単位)。

詳細については、「コンポーネントのシステムリソース制限を設定する」を参照してください。

この機能は、Linux コアデバイス上の Greengrass nucleus コンポーネントの v2.4.0 以降および Greengrass CLI で使用できます。 AWS IoT Greengrass 現在、Windows コアデバイスではこの機能をサポートしていません。

- `--remove`。ローカルデプロイから削除するターゲットコンポーネントの名前。クラウドデプロイからマージされたコンポーネントを削除するには、ターゲットモノグループのグループ ID を次の形式で指定する必要があります。

  Greengrass nucleus v2.4.0 and later

  ```
  --remove <component-name> --groupId <group-name>
  ```

  Earlier than v2.4.0

  ```
  --remove <component-name> --groupId thinggroup/<group-name>
  ```

- `--failure-handling-policy`。デプロイが失敗したときに実行されるアクションを定義します。指定できるアクションは次の 2 つです。
  - `ROLLBACK` –
  - `DO_NOTHING` –

  この機能は Greengrass nucleus の v2.11.0 以降で使用できます。

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。

```
$ sudo greengrass-cli deployment create \
    --merge MyApp1=1.0.0 \
    --merge MyApp2=1.0.0 --runWith MyApp2:posixUser=ggc_user \
    --remove MyApp3 \
    --recipeDir recipes/ \
    --artifactDir artifacts/

Local deployment has been submitted! Deployment Id: 44d89f46-1a29-4044-
ad89-5151213dfcbc
```

キャンセル

指定されたデプロイをキャンセルします。

概要

```
greengrass-cli deployment cancel
    -i <deployment-id>
```

引数

`-i`。キャンセルするデプロイの一意の識別子。デプロイ ID は `create` コマンドの出力で返されます。

出力

- なし

list

過去 10 回分のローカルデプロイのステータスを取得します。

概要

```
greengrass-cli deployment list
```

引数

なし

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。デプロイのステータスに応じて、出力には IN_PROGRESS、SUCCEEDED、または FAILED のいずれかのステータス値が表示されます。

```
$ sudo greengrass-cli deployment list

44d89f46-1a29-4044-ad89-5151213dfcbc: SUCCEEDED
Created on: 6/27/23 11:05 AM
```

status

特定のデプロイのステータスを取得します。

概要

```
greengrass-cli deployment status -i <deployment-id>
```

引数

-i。 デプロイの ID。

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。デプロイのステータスに応じて、出力には IN_PROGRESS、SUCCEEDED、または FAILED のいずれかのステータス値が表示されます。

```
$ sudo greengrass-cli deployment status -i 44d89f46-1a29-4044-ad89-5151213dfcbc

44d89f46-1a29-4044-ad89-5151213dfcbc: FAILED
Created on: 6/27/23 11:05 AM
Detailed Status: <Detailed deployment status>
```

```
Deployment Error Stack: List of error codes
Deployment Error Types: List of error types
Failure Cause: Cause
```

ログ

logs コマンドを使用して、コアデバイスの Greengrass ログを分析します。

サブコマンド

- get
- list-keywords
- list-log-files

get

Greengrass ログファイルを収集、フィルタリング、視覚化します。このコマンドは JSON 形式のログファイルのみをサポートします。nucleus 設定のログ記録形式を指定できます。

概要

```
greengrass-cli logs get
    [--log-dir path/to/a/log/folder]
    [--log-file path/to/a/log/file]
    [--follow true | false ]
    [--filter <filter> ]
    [--time-window <start-time>,<end-time> ]
    [--verbose ]
    [--no-color ]
    [--before <value> ]
    [--after <value> ]
    [--syslog ]
    [--max-long-queue-size <value> ]
```

引数

- --log-dir、-ld。ログファイルをチェックするディレクトリのパス (例: ***/greengrass/v2/logs***)。--syslog と併用しないでください。指定する追加ディレクトリに個別の引数を使用します。少なくくとも --log-dir、--log-file の内いずれかを指定する必要があります。1 つのコマンドで 両方の引数を使用することもできます。

- `--log-file`、`-lf`。使用するログディレクトリへのパス。指定する追加ディレクトリに個別の引数を使用します。少なくくとも `--log-dir`、`--log-file` の内いずれかを指定する必要があります。1 つのコマンドで 両方の引数を使用することもできます。
- `--follow`、`-fol`。発生するに従って、ログの更新を表示します。Greengrass CLI は引き続き実行され、指定されたログから読み取られます。時間ウィンドウを指定している場合、Greengrass CLI はすべての時間ウィンドウが終了した後にログのモニタリングを停止します。
- `--filter`、`-f`。フィルターとして使用するキーワード、正規表現、またはキーと値のペア。この値は、文字列、正規表現、またはキーと値のペアとして指定します。指定する追加フィルターごとに個別の引数を使用します。

  評価後、単一の引数内に指定されている複数のフィルターは OR 演算子で区切られ、追加の引数で指定されたフィルターは AND 演算子で結合されます。たとえば、コマンドに `--filter "installed" --filter "name=alpha,name=beta"` を含めた場合、Greengrass CLI は、`alpha` または `beta` を値に持つ `installed` キーワードと `name` キーの両方が含まれるログメッセージをフィルタリングして表示します。
- `--time-window`、`-t`。ログ情報を表示する時間ウィンドウ。正確なタイムスタンプと相対オフセットの両方を使用できます。*`<begin-time>,<end-time>`* 形式でこの情報を提供する必要があります。開始時間または終了時間を指定しなかった場合、そのオプションの値はデフォルトで、現在のシステムの日付と時刻に設定されます。指定する追加時間ウィンドウごとに個別の引数を使用します。

  Greengrass CLI では、次の形式のタイムスタンプがサポートされています。

  - `yyyy-MM-DD`、例えば `2020-06-30` など。この形式を使用すると、時間のデフォルトは 00:00:00 になります。

    `yyyyMMDD`、例えば `20200630` など。この形式を使用すると、時間のデフォルトは 00:00:00 になります。

    `HH:mm:ss`、例えば `15:30:45` など。この形式を使用すると、日付は現在のシステム日付にデフォルト設定されます。

    `HH:mm:ssSSS`、例えば `15:30:45` など。この形式を使用すると、日付は現在のシステム日付にデフォルト設定されます。

    `YYYY-MM-DD'T'HH:mm:ss'Z'`、例えば `2020-06-30T15:30:45Z` など。

    `YYYY-MM-DD'T'HH:mm:ss`、例えば `2020-06-30T15:30:45` など。

yyyy-MM-dd'T'HH:mm:ss.SSS、例えば 2020-06-30T15:30:45.250 など。

相対オフセットは、現在のシステム時刻からの時間オフセットを指定します。Greengrass CLI は、相対オフセットに次の形式をサポートしています：+|-[<*value*>h|hr|hours][*value*m|min|minutes][*value*]s|sec|seconds。

例えば、現在時刻の 1 時間前から 2 時間 15 分前までの時間帯を指定する以下の引数は、--time-window -2h15min,-1hr です。

- --verbose。 ログメッセージのすべてのフィールドを表示します。--syslog と併用しないでください。
- --no-color、-nc。カラーコーディングを削除します。ログメッセージのデフォルトのカラーコードでは、太字の赤いテキストが使用されます。ANSI エスケープシーケンスを使用するため、UNIX 互換の端末のみをサポートします。
- --before、-b。一致したログエントリの前に表示する行数。デフォルトは0です。
- --after、-a。一致したログエントリの後に表示する行数。デフォルトは0です。
- --syslog。RFC3164 で定義された syslog プロトコルを使用して、すべてのログファイルを処理します。--log-dir および--verbose と併用しないでください。syslog プロトコルでは、次の形式を使用します: "<$Priority>$Timestamp $Host $Logger ($Class): $Message"。ログファイルを指定しなかった場合、Greengrass CLI は次の場所からログメッセージを読み取ります: /var/log/messages、/var/log/syslog、または/var/log/system.log。

  AWS IoT Greengrass は、現在 Windows コアデバイスにこの機能をサポートしていません。
- --max-log-queue-size、-m。メモリに割り当てるログエントリの最大数。このオプションを使用して、メモリ使用量を最適化します。デフォルトは 100 です。

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。

```
$ sudo greengrass-cli logs get --verbose \
    --log-file /greengrass/v2/logs/greengrass.log \
    --filter deployment,serviceName=DeploymentService \
    --filter level=INFO \
    --time-window 2020-12-08T01:11:17,2020-12-08T01:11:22

2020-12-08T01:11:17.615Z [INFO] (pool-2-thread-14)
 com.aws.greengrass.deployment.DeploymentService: Current deployment finished.
```

```
 {DeploymentId=44d89f46-1a29-4044-ad89-5151213dfcbc, serviceName=DeploymentService,
 currentState=RUNNING}
2020-12-08T01:11:17.675Z [INFO] (pool-2-thread-14)
 com.aws.greengrass.deployment.IotJobsHelper: Updating status of persisted
 deployment. {Status=SUCCEEDED, StatusDetails={detailed-deployment-
status=SUCCESSFUL}, ThingName=MyThing, JobId=22d89f46-1a29-4044-ad89-5151213dfcbc
```

list-keywords

ログファイルのフィルタリングに使用できる推奨キーワードを表示します。

概要

```
greengrass-cli logs list-keywords [arguments]
```

引数

なし

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。

```
$ sudo greengrass-cli logs list-keywords

Here is a list of suggested keywords for Greengrass log:
level=$str
thread=$str
loggerName=$str
eventType=$str
serviceName=$str
error=$str
```

```
$ sudo greengrass-cli logs list-keywords --syslog

Here is a list of suggested keywords for syslog:
priority=$int
host=$str
logger=$str
class=$str
```

list-log-files

指定したディレクトリにあるログファイルを表示します。

概要

```
greengrass-cli logs list-log-files [arguments]
```

引数

`--log-dir`、`-ld`。ログファイルをチェックするディレクトリのパス。

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。

```
$ sudo greengrass-cli logs list-log-files -ld /greengrass/v2/logs/

/greengrass/v2/logs/aws.greengrass.Nucleus.log
/greengrass/v2/logs/main.log
/greengrass/v2/logs/greengrass.log
Total 3 files found.
```

get-debug-password

`get-debug-password` コマンドを使用すると、ローカルデバッグコンソールコンポーネント (`aws.greengrass.LocalDebugConsole`) 向けにランダムに生成されたパスワードが出力されます。パスワードは、生成されてから 8 時間後に期限切れになります。

概要

```
greengrass-cli get-debug-password
```

引数

なし

出力

次の例は、このコマンドを実行したときに生成される出力を示しています。

```
$ sudo greengrass-cli get-debug-password
```

```
Username: debug
Password: bEDp3MOHdj8ou2w5de_sCBI2XAaguy3a8XxREXAMPLE
Password expires at: 2021-04-01T17:01:43.921999931-07:00
The local debug console is configured to use TLS security. The certificate is self-
signed so you will need to bypass your web browser's security warnings to open the
 console.
Before you bypass the security warning, verify that the certificate fingerprint
 matches the following fingerprints.
SHA-256: 15 0B 2C E2 54 8B 22 DE 08 46 54 8A B1 2B 25 DE FB 02 7D 01 4E 4A 56 67 96
 DA A6 CC B1 D2 C4 1B
SHA-1: BC 3E 16 04 D3 80 70 DA E0 47 25 F9 90 FA D6 02 80 3E B5 C1
```

AWS IoT Greengrass テストフレームワークを使用する

Greengrass Testing Framework (GTF) は、顧客の観点からオートメーションをサポートする end-to-end構成要素のコレクションです。GTF は Cucumber を特徴量ドライバーとして使用します。 は、さまざまなデバイスでソフトウェアの変更を認定するために、同じ構成要素 AWS IoT Greengrass を使用します。詳細については、Github の「Greengrass テストフレームワーク」を参照してください。

GTF は、コンポーネントの動作主導型開発 (BDD) を促進するために、自動テストを実行するツールである Cucumber を使用して実装されています。Cucumber では、このシステムの機能の概要は、`feature` という特殊なタイプのファイルにまとめられています。各機能は、自動テストに変換できる仕様を持つ、シナリオと呼ばれる人間が読める形式で記述されています。各シナリオは、Gherkin と呼ばれるドメイン固有の言語を使用して、テスト対象のシステムの相互作用と結果を定義する一連のステップとして概説されています。Gherkin ステップは、仕様をテストフローに固定するステップ定義と呼ばれる方法を使用してプログラミングコードにリンクされます。GTF のステップ定義は Java で実装されます。

トピック

- 仕組み
- 変更ログ
- Greengrass テストフレームワークの設定オプション
- チュートリアル: Greengrass Testing Framework と Greengrass Development Kit を使用して end-to-end テストを実行する
- チュートリアル: 信頼テストスイートからの信頼テストを使用する

仕組み

AWS IoT Greengrass は、GTF を複数の Java モジュールで構成されるスタンドアロン JAR として配布します。GTF を使用してコンポーネントの end-to-end テストを行うには、Java プロジェクト内でテストを実装する必要があります。テスト用スタンドタブル JAR を Java プロジェクトの依存関係として追加すると、GTF の既存の機能を使用したり、独自のカスタムテストケースを作成することで拡張したりできるようになります。カスタムテストケースを実行するには、Java プロジェクトを構築し、Greengrass テストフレームワークの設定オプション で説明されている設定オプションを使用してターゲット JAR を実行します。

GTF スタンドアロン JAR

Greengrass は Cloudfront を Maven リポジトリとして使用し、GTF スタンドアロン JAR の異なるバージョンをホストしています。GTF バージョンの全リストについては、GTF リリースを参照してください。

GTF スタンドアロン JAR には以下のモジュールが含まれています。これらのモジュールだけに限定されるものではありません。これらの依存関係をプロジェクト内で個別に選択することも、テスト用スタンドアロン JAR ファイルですべての依存関係をプロジェクトに一度に含めることもできます。

- `aws-greengrass-testing-resources`: このモジュールは、テストの過程で AWS リソースのライフサイクルを管理するための抽象化を提供します。これを使用して抽`ResourceSpec`象化を使用してカスタム AWS リソースを定義し、GTF がそれらのリソースの作成と削除を自動的に処理できるようにします。
- `aws-greengrass-testing-platform`: このモジュールは、テストライフサイクル中にテスト対象のデバイスをプラットフォームレベルで抽象化します。プラットフォームとは独立して OS とやりとりするための API が含まれており、デバイスシェルで実行されるコマンドをシミュレートするために使用できます。
- `aws-greengrass-testing-components`: このモジュールは、デプロイメント、IPC、その他の機能などの Greengrass のコア機能のテストに使用されるサンプルコンポーネントで構成されています。
- `aws-greengrass-testing-features`: このモジュールは、Greengrass 環境でのテストに使用される再利用可能な共通ステップとその定義で構成されています。

トピック

- 変更ログ
- Greengrass テストフレームワークの設定オプション

- チュートリアル: Greengrass Testing Framework と Greengrass Development Kit を使用して end-to-end テストを実行する
- チュートリアル: 信頼テストスイートからの信頼テストを使用する

変更ログ

次の表に、GTF の各バージョンの変更を示します。詳細については、「」の「GTF リリース」ページを参照してください GitHub。

バージョン	変更
1.2.0	新機能 • テスト中に MQTT とインターネット接続を設定するためのネットワーク関連のステップを追加しました。 • デバイス RAM と CPU 使用率をモニタリングするためのシステムメトリクスステップが追加されました。 バグ修正と機能向上 • Greengrass CLI ローカルデプロイステップは、成功するまで再試行します。 • Greengrass nucleus を強制終了するのではなく、適切に停止するテスト。 • モノとロールエイリアスに対して AWS IoT 認証情報を取得するまで、GTF が認証情報エンドポイントをポーリングする機能が向上しました。 • 欠落しているアーティファクトと recipe ディレクトリを修正しました。このバージョンでは、欠落しているコンポーネントバージョンも修正されています。 • Docker イメージが存在しない場合、Docker イメージのクリーンアップ中に GTF が失敗する問題を修正しました。 • コンポーネントのバージョンとして CURRENT キーワードを追加します。

バージョン	変更
1.1.0	新機能 • 設定を使用してカスタムコンポーネントをインストールする機能が追加されました。これには、カスタムコンポーネントのレシピが必要です。 • カスタム設定でローカルデプロイを更新する機能が追加されました。 バグ修正と機能向上 • ログコンテキストの GTF バージョンの不整合の問題を修正しました。
1.0.0	当初のバージョン

Greengrass テストフレームワークの設定オプション

GTF の設定オプション

Greengrass テストフレームワーク (GTF) では、エンドツーエンドのテストプロセスの開始時に特定のパラメータを設定して、テストフローを調整できます。これらの設定オプションは GTF スタンドアロン JAR の CLI 引数として指定できます。

GTF バージョン 1.1.0 以降には、以下の設定オプションがあります。

- `additional-plugins` – (オプション) 追加の Cucumber プラグイン
- `aws-region` – AWS のサービスの特定のリージョンレベルのエンドポイントをターゲットにします。デフォルトは、AWS SDK が検出するものです。
- `credentials-path` – オプションの AWS プロファイル認証情報のパス。デフォルトは、ホスト環境で検出された認証情報です。
- `credentials-path-rotation` – AWS 認証情報のオプションのローテーション期間。デフォルトで 15 分または `PT15M` に設定されます。
- `csr-path` – デバイス証明書の生成に使用される CSR のパス。
- `device-mode` – テスト対象のターゲットデバイス。デフォルトはローカルデバイスです。
- `env-stage` – Greengrass のデプロイ環境をターゲットに設定します。デフォルトは本番です。
- `existing-device-cert-arn` – Greengrass のデバイス証明書として使用する既存の証明書の ARN。
- `feature-path` – 追加の機能ファイルを含むファイルまたはディレクトリ。デフォルトでは、追加の機能ファイルは使用されません。

- gg-cli-version – Greengrass CLI のバージョンをオーバーライドします。デフォルトは ggc.version にある値です。
- gg-component-bucket – Greengrass コンポーネントを格納する既存の Amazon S3 バケットの名前。
- gg-component-overrides – Greengrass コンポーネントのオーバーライドのリスト。
- gg-persist – テスト実行後に保持されるテスト要素のリスト。デフォルトでは、何も保持しないよう設定されています。許容される値は、aws.resources、installed.software、および generated.files です。
- gg-runtime – テストがテストリソースとどのようにインタラクションするかに影響する値のリスト。これらの値は gg.persist パラメータよりも優先されます。デフォルトが空の場合、インストールされている Greengrass ランタイムを含む、すべてのテストリソースがテストケースによって管理されると想定されます。許容される値は、aws.resources、installed.software、および generated.files です。
- ggc-archive – アーカイブされた Greengrass nucleus コンポーネントへのパス。
- ggc-install-root – Greengrass nucleus コンポーネントをインストールするディレクトリ。デフォルトは test.temp.path とテスト実行フォルダです。
- ggc-log-level – テスト実行の Greengrass nucleus ログレベルを設定します。デフォルトは「INFO」です。
- ggc-tes-rolename – AWS のサービスにアクセスするために AWS IoT Greengrass Core が引き受ける IAM ロール。指定された名前のロールが存在しない場合は、ロールとデフォルトのアクセスポリシーが作成されます。
- ggc-trusted-plugins – Greengrass に追加する必要がある、信頼されたプラグインのパス (ホスト上) のカンマ区切りリスト。DUT 自体のパスを指定するには、パスの前に「dut:」というプレフィックスを付けます。
- ggc-user-name – Greengrass nucleus の user:group posixUser の値。デフォルトは、ログインしている現在のユーザー名です。
- ggc-version – 実行中の Greengrass nucleus コンポーネントのバージョンをオーバーライドします。デフォルトは ggc.archive にある値です。
- log-level – テスト実行のログレベル。デフォルトは「INFO」です。
- parallel-config – JSON 文字列としてのバッチインデックスとバッチ数のセット。バッチインデックスのデフォルト値は 0、バッチ数は 1 です。
- proxy-url – この URL を介してトラフィックをルーティングするようにすべてのテストを設定します。

- `tags` – 機能タグのみを実行します。「&」を使用して組み合わせることができます
- `test-id-prefix` – AWS リソース名とタグを含むすべてのテスト固有のリソースに適用される共通のプレフィックス。デフォルトは「gg」プレフィックスです。
- `test-log-path` – テスト実行全体の結果を含むディレクトリ。デフォルトは「testResults」です。
- `test-results-json` – 結果として得られる Cucumber JSON レポートがディスクに書き込まれた状態で生成されるかどうかを決定するフラグ。デフォルトは true です。
- `test-results-log` – コンソール出力がディスクに書き込まれた状態で生成されるかどうかを決定するフラグ。デフォルトは false です。
- `test-results-xml` – 結果として得られる JUnit XML レポートがディスクに書き込まれた状態で生成されるかどうかを決定するフラグ。デフォルトは true です。
- `test-temp-path` – ローカルテストアーティファクトを生成するディレクトリ。デフォルトは、gg-testing というプレフィックスが付いたランダムな一時ディレクトリです。
- `timeout-multiplier` – すべてのテストタイムアウトに提供される乗数。デフォルトは 1.0 です。

チュートリアル: Greengrass Testing Framework と Greengrass Development Kit を使用して end-to-end テストを実行する

AWS IoT Greengrass テストフレームワーク (GTF) と Greengrass Development Kit (GDK) は、デベロッパーが end-to-end テストを実行する方法を提供します。このチュートリアルを完了すると、 コンポーネントで GDK プロジェクトを初期化し、 end-to-end テストモジュールで GDK プロジェクトを初期化し、カスタムテストケースを構築できます。カスタムテストケースの作成後、テストを実行できます。

このチュートリアルでは、以下の作業を行います。

1. GDK プロジェクトをコンポーネントで初期化します。
2. end-to-end テストモジュールを使用して GDK プロジェクトを初期化します。
3. カスタムテストケースを構築します。
4. 新しいテストケースにタグを追加します。
5. テスト JAR をビルドします。
6. テストを実行します。

トピック

前提条件

このチュートリアルを完了するには、以下が必要です。

- GDK バージョン 1.3.0 以降
- Java
- Maven
- Git

ステップ 1: GDK プロジェクトをコンポーネントで初期化する

- GDK プロジェクトで空のフォルダを初期化します。以下のコマンドを実行して Python で実装された `HelloWorld` コンポーネントをダウンロードします。

  ```
  gdk component init -t HelloWorld -l python -n HelloWorld
  ```

  このコマンドで、現在のディレクトリに `HelloWorld` という名前の新しいディレクトリが作成されます。

ステップ 2: テストモジュールを使用して GDK プロジェクトを初期化する end-to-end

- GDK を使用して、テストモジュールテンプレートをダウンロードできます。このテンプレートには、特定の機能とステップの実装が含まれています。以下のコマンドを実行して `HelloWorld` ディレクトリを開き、テストモジュールを使用して既存の GDK プロジェクトを初期化します。

```
cd HelloWorld
gdk test-e2e init
```

このコマンドで、HelloWorld ディレクトリに gg-e2e-tests という名前の新しいディレクトリが作成されます。このテストディレクトリは Greengrass のテスト用スタンドアロン JAR に依存する Maven プロジェクトです。

ステップ 3 : カスタムテストケースを構築する

カスタムテストケースの作成は、大きく分けて 2 つのステップで構成されます。テストシナリオを持つ機能ファイルを作成することと、もう 1 つはステップ定義を実装することです。カスタムテストケースの作成例については、例: カスタムテストケースを構築する を参照してください。以下の手順でカスタムテストケースを構築してください。

1. テストシナリオを持つ機能ファイルを作成する

   機能は通常、テスト対象のソフトウェアの特定の機能を表します。Cucumber では、各機能はタイトル、詳細な説明、シナリオと呼ばれる特定のケースの 1 つまたは複数の例を含む個別の機能ファイルとして指定されます。各シナリオは、タイトル、詳細な説明、相互作用と期待される結果を定義する一連のステップで構成されています。シナリオは、「given」、「when」、「then」というキーワードを使用して構造化された形式で記述されます。

2. ステップ定義を実装する

   ステップ定義は、プログラマティックコードに Gherkin ステップ をプレインランゲージでリンクします。Cucumber はシナリオ内の Gherkin ステップを識別すると、一致するステップ定義を探して実行します。

ステップ 4 : 新しいテストケースにタグを追加する

- 機能やシナリオにタグを割り当てて、テストプロセスを整理できます。タグを使用してシナリオのサブセットを分類したり、実行するフックを条件付きで選択したりできます。機能とシナリオには、スペースで区切ることで複数のタグを付けることができます。

  この例では、HelloWorld コンポーネントを使用します。

  機能ファイルに、@Sample タグの横に @HelloWorld という名前の新しいタグを追加します。

```
@Sample @HelloWorld
Scenario: As a developer, I can create a component and deploy it on my device
....
```

ステップ 5: テスト JAR を構築する

1. コンポーネントを構築します。テストモジュールを構築する前に、コンポーネントを構築する必要があります。

```
gdk component build
```

2. 次のコマンドを使用してテストモジュールを構築します。このコマンドは、`greengrass-build` フォルダーにテスト用 JAR を構築します。

```
gdk test-e2e build
```

ステップ 6 : テストを実行する

カスタムテストケースを実行すると、GTF はテスト中に作成されたリソースを管理するとともに、テストのライフサイクルを自動化します。まず、テスト対象デバイス (DUT) を AWS IoT 機器としてプロビジョニングし、そのデバイスに Greengrass コアソフトウェアをインストールします。次に、そのパスで指定されたレシピを使用して `HelloWorld` という名前の新しいコンポーネントを作成します。その後、`HelloWorld`コンポーネントは Greengrass モノのデプロイを介してコアデバイスにデプロイされます。その後、デプロイが成功したかどうかが検証されます。デプロイが成功すると 3 分以内にデプロイステータスが `COMPLETED` に変わります。

1. プロジェクトディレクトリ内の `gdk-config.json` ファイルに移動し、`HelloWorld` タグを持つテストをターゲットにします。次のコマンドを使用して、`test-e2e` キーをアップデートします。

```
 "test-e2e":{
   "gtf_options" : {
        "tags":"HelloWorld"
    }
 }
```

2. テストを実行する前に、AWS ホストデバイスへの認証情報を入力する必要があります。GTF はこれらの認証情報を使用してテストプロセス中の AWS リソースを管理します。指定したロールに、テストに含まれる必要な操作を自動化する権限があることを確認してください。

   次のコマンドを実行して、AWS 認証情報を提供します。

   - Linux or Unix

     ```
     export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
     export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
     ```

     Windows Command Prompt (CMD)

     ```
     set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
     set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
     ```

     PowerShell

     ```
     $env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
     $env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
     ```

3. 次のコマンドを使用してテストを実行します。

   ```
   gdk test-e2e run
   ```

   このコマンドは、`greengrass-build` フォルダーにある Greengrass nucleus の最新バージョンをダウンロードし、それを使用してテストを実行します。また、このコマンドは、`HelloWorld` タグを持つシナリオのみをターゲットにし、それらのシナリオのレポートを生成します。このテスト中に作成されたAWSリソースは、テストの最後に破棄されます。

例: カスタムテストケースを構築する

Example

GDK プロジェクトにダウンロードされたテストモジュールは、サンプル機能とステップ実装ファイルで構成されています。

以下の例では、Greengrass ソフトウェアのデバイスデプロイ機能をテストするための機能ファイルを作成します。Greengrass AWS クラウド を介してコンポーネントをデプロイするシナリオを使用

して、この機能の機能を部分的にテストします。これは、このユースケースの相互作用と期待される結果を理解するのに役立つ一連のステップです。

1. 機能ファイルを作成する

   現在のディレクトリ内のgg-e2e-tests/src/main/resources/greengrass/featuresフォルダに移動します。次の例component.featureのようなサンプルを見つけることができます。

   この機能ファイルでは、Greengrass ソフトウェアのデバイスデプロイ機能をテストできます。Greengrass クラウドを介したコンポーネントのデプロイを実行するシナリオで、この機能の機能を部分的にテストできます。このシナリオは、このユースケースの相互作用と期待される結果を理解するのに役立つ一連のステップです。

   ```
   Feature: Testing features of Greengrassv2 component

   Background:
       Given my device is registered as a Thing
       And my device is running Greengrass

   @Sample
   Scenario: As a developer, I can create a component and deploy it on my device
       When I create a Greengrass deployment with components
           HelloWorld | /path/to/recipe/file
       And I deploy the Greengrass deployment configuration
       Then the Greengrass deployment is COMPLETED on the device after 180 seconds
       And I call my custom step
   ```

   GTF には、And I call my custom step という名前のステップを除く以下のすべてのステップのステップ定義が含まれています。

2. ステップ定義を実装する

   GTF スタンドアロン JAR には、And I call my custom step ステップを除くすべてのステップのステップ定義が含まれています。このステップをテストモジュールに実装できます。

   テストファイルのソースコードに移動します。以下のコマンドを使用すると、ステップ定義を使用してカスタムステップをリンクできます。

   ```
   @And("I call my custom step")
   ```

```
public void customStep() {
    System.out.println("My custom step was called ");
}
```

チュートリアル: 信頼テストスイートからの信頼テストを使用する

AWS IoT Greengrass テストフレームワーク (GTF) と Greengrass Development Kit (GDK) は、デベロッパーが end-to-end テストを実行する方法を提供します。このチュートリアルを完了すると、 コンポーネントで GDK プロジェクトを初期化し、 end-to-end テストモジュールで GDK プロジェクトを初期化し、 信頼度テストスイートからの信頼テストを使用できます。カスタムテストケースの作成後、テストを実行できます。

信頼テストは、Greengrass が提供する一般的なテストで、基本的なコンポーネントの動作を検証します。これらのテストは、より具体的なコンポーネントニーズに合わせて変更または拡張できます。

このチュートリアルでは、 HelloWorld コンポーネントを使用します。別のコンポーネントを使用している場合は、 HelloWorld コンポーネントをコンポーネントに置き換えます。

このチュートリアルでは、以下の作業を行います。

1. GDK プロジェクトをコンポーネントで初期化します。
2. end-to-end テストモジュールを使用して GDK プロジェクトを初期化します。
3. 信頼度テストスイートのテストを使用します。
4. 新しいテストケースにタグを追加します。
5. テスト JAR をビルドします。
6. テストを実行します。

トピック

- 例: 信頼テストを使用する

前提条件

このチュートリアルを完了するには、以下が必要です。

- GDK バージョン 1.6.0 以降
- Java
- Maven
- Git

ステップ 1: GDK プロジェクトをコンポーネントで初期化する

- GDK プロジェクトで空のフォルダを初期化します。以下のコマンドを実行して Python で実装された `HelloWorld` コンポーネントをダウンロードします。

  ```
  gdk component init -t HelloWorld -l python -n HelloWorld
  ```

  このコマンドで、現在のディレクトリに `HelloWorld` という名前の新しいディレクトリが作成されます。

ステップ 2: テストモジュールを使用して GDK プロジェクトを初期化する end-to-end

- GDK を使用して、テストモジュールテンプレートをダウンロードできます。このテンプレートには、特定の機能とステップの実装が含まれています。以下のコマンドを実行して `HelloWorld` ディレクトリを開き、テストモジュールを使用して既存の GDK プロジェクトを初期化します。

  ```
  cd HelloWorld
  gdk test-e2e init
  ```

  このコマンドで、`HelloWorld` ディレクトリに `gg-e2e-tests` という名前の新しいディレクトリが作成されます。このテストディレクトリは Greengrass のテスト用スタンドアロン JAR に依存する Maven プロジェクトです。

ステップ 3: 信頼度テストスイートのテストを使用する

信頼度テストケースの作成は、提供された機能ファイルの使用と、必要に応じてシナリオの変更で構成されます。信頼テストの使用例については、「」を参照してください例: カスタムテストケースを構築する。信頼度テストを使用するには、次の手順に従います。

- 提供された機能ファイルを使用します。

  現在のディレクトリ内のgg-e2e-tests/src/main/resources/greengrass/featuresフォルダに移動します。サンプルconfidenceTest.featureファイルを開き、信頼テストを使用します。

ステップ 4 : 新しいテストケースにタグを追加する

- 機能やシナリオにタグを割り当てて、テストプロセスを整理できます。タグを使用してシナリオのサブセットを分類したり、実行するフックを条件付きで選択したりできます。機能とシナリオには、スペースで区切ることで複数のタグを付けることができます。

  この例では、HelloWorld コンポーネントを使用します。

  各シナリオには のタグが付けられます@ConfidenceTest。テストスイートのサブセットのみを実行する場合は、タグを変更または追加します。各テストシナリオは、各信頼テストの最上部で説明されています。このシナリオは、各テストケースのインタラクションと期待される結果を理解するのに役立つ一連のステップです。これらのテストを拡張するには、独自のステップを追加するか、既存のステップを変更します。

  ```
  @ConfidenceTest
  Scenario: As a Developer, I can deploy GDK_COMPONENT_NAME to my device and see it
   is working as expected
  ....
  ```

ステップ 5: テスト JAR を構築する

1. コンポーネントを構築します。テストモジュールを構築する前に、コンポーネントを構築する必要があります。

   ```
   gdk component build
   ```

2. 次のコマンドを使用してテストモジュールを構築します。このコマンドは、greengrass-build フォルダーにテスト用 JAR を構築します。

```
gdk test-e2e build
```

ステップ 6 : テストを実行する

信頼度テストを実行すると、GTF はテスト中に作成されたリソースの管理とともにテストのライフサイクルを自動化します。まず、テスト対象デバイス (DUT) を AWS IoT 機器としてプロビジョニングし、そのデバイスに Greengrass コアソフトウェアをインストールします。次に、そのパスで指定されたレシピを使用して HelloWorld という名前の新しいコンポーネントを作成します。その後、HelloWorldコンポーネントは Greengrass モノのデプロイを介してコアデバイスにデプロイされます。その後、デプロイが成功したかどうかが検証されます。デプロイが成功すると 3 分以内にデプロイステータスが COMPLETED に変わります。

1. プロジェクトディレクトリの gdk-config.json ファイルに移動して、ステップ 4 で指定した ConfidenceTest タグまたはタグ TAK8u を持つテストをターゲットにします。次のコマンドを使用して、test-e2e キーをアップデートします。

```
"test-e2e":{
  "gtf_options" : {
       "tags":"ConfidenceTest"
   }
}
```

2. テストを実行する前に、AWS ホストデバイスへの認証情報を入力する必要があります。GTF はこれらの認証情報を使用してテストプロセス中の AWS リソースを管理します。指定したロールに、テストに含まれる必要な操作を自動化する権限があることを確認してください。

   次のコマンドを実行して、AWS 認証情報を提供します。

   - Linux or Unix

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

   Windows Command Prompt (CMD)

```
set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
```

```
set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

PowerShell

```
$env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
$env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
```

3. 次のコマンドを使用してテストを実行します。

```
gdk test-e2e run
```

このコマンドは、greengrass-build フォルダーにある Greengrass nucleus の最新バージョンをダウンロードし、それを使用してテストを実行します。また、このコマンドは、ConfidenceTest タグを持つシナリオのみをターゲットにし、それらのシナリオのレポートを生成します。このテスト中に作成されたAWSリソースは、テストの最後に破棄されます。

例: 信頼テストを使用する

Example

GDK プロジェクトでダウンロードされたテストモジュールは、提供された機能ファイルで構成されています。

次の例では、Greengrass ソフトウェアのモノのデプロイ機能をテストするための機能ファイルを使用します。Greengrass AWS クラウド を介してコンポーネントをデプロイするシナリオを使用して、この機能の機能を部分的にテストします。これは、このユースケースの相互作用と期待される結果を理解するのに役立つ一連のステップです。

- 提供された機能ファイルを使用します。

  現在のディレクトリ内のgg-e2e-tests/src/main/resources/greengrass/featuresフォルダに移動します。次の例confidenceTest.featureのようなサンプルを見つけることができます。

```
Feature: Confidence Test Suite

Background:
    Given my device is registered as a Thing
    And my device is running Greengrass
```

```
@ConfidenceTest
Scenario: As a Developer, I can deploy GDK_COMPONENT_NAME to my device and see it
 is working as expected
    When I create a Greengrass deployment with components
      | GDK_COMPONENT_NAME | GDK_COMPONENT_RECIPE_FILE |
      | aws.greengrass.Cli | LATEST                    |
    And I deploy the Greengrass deployment configuration
    Then the Greengrass deployment is COMPLETED on the device after 180 seconds
    # Update component state accordingly. Possible states: {RUNNING, FINISHED,
 BROKEN, STOPPING}
    And I verify the GDK_COMPONENT_NAME component is RUNNING using the greengrass-
cli
```

各テストシナリオは、各信頼テストの最上部で説明されています。このシナリオは、各テストケースのインタラクションと期待される結果を理解するのに役立つ一連のステップです。これらのテストを拡張するには、独自のステップを追加するか、既存のステップを変更します。各シナリオには、これらの調整に役立つコメントが含まれています。

AWS IoT Greengrass コンポーネントを開発する

Greengrass コアデバイスでコンポーネントの開発とテストを行うことができます。そのため、AWS クラウド と対話することなく、AWS IoT Greengrass ソフトウェアを作成して繰り返すことが可能です。コンポーネントのバージョンが完成したら、クラウドの AWS IoT Greengrass にアップロードすることができるため、自分と自分が所属するチームが、コンポーネントをフリート内の他のデバイスにデプロイすることができます。コンポーネントをデプロイする方法の詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

すべてのコンポーネントは、レシピとアーティファクトで設定されます。

- recipe

  すべてのコンポーネントには、メタデータを定義する recipe ファイルが含まれています。recipe では、コンポーネントの設定パラメータ、コンポーネントの依存関係、ライフサイクル、プラットフォームの互換性も指定します。コンポーネントのライフサイクルは、コンポーネントのインストール、実行、およびシャットダウンを行うコマンドを定義します。詳細については、「AWS IoT Greengrass コンポーネントレシピのリファレンス」を参照してください。

  recipe は JSON または YAML 形式で定義できます。

- アーティファクト

コンポーネントは、コンポーネントバイナリであるアーティファクトを必要な数だけ持つことができます。アーティファクトには、スクリプト、コンパイルされたコード、静的リソース、およびコンポーネントが消費するその他のファイルが含まれます。コンポーネントはコンポーネントの依存関係からアーティファクトを消費することもできます。

AWS IoT Greengrass では、アプリケーションで使用して、デバイスにデプロイすることができる事前に構築済みのコンポーネントが提供されています。例えば、ストリームマネージャーコンポーネントを使用してさまざまなAWSサービスにデータをアップロードしたり、 CloudWatch メトリクスコンポーネントを使用してカスタムメトリクスを Amazon に発行したりできます CloudWatch。詳細については、「AWS が提供したコンポーネント」を参照してください。

AWS IoT Greengrass は Greengrass ソフトウェアカタログと呼ばれる Greengrass コンポーネントのインデックスをキュレーションします。このカタログは、Greengrass コミュニティによって開発された Greengrass コンポーネントを追跡します。このカタログから、コンポーネントをダウンロード、変更、デプロイして Greengrass アプリケーションを作成できます。詳細については、「コミュニティコンポーネント」を参照してください。

AWS IoT Greengrass Core ソフトウェアは、コアデバイスで設定した `ggc_user` や `ggc_group` などのコンポーネントをシステムユーザーおよびグループとして実行します。これは、コンポーネントがそのシステムユーザーの権限を持っていることを意味します。ホームディレクトリを持たないシステムユーザーを使用した場合、コンポーネントはホームディレクトリを使用する実行コマンドやコードを使用できません。これは、Python パッケージをインストールする場合などに、`pip install some-library --user` コマンドを使用できないことを意味します。入門チュートリアルに従ってコアデバイスを設定している場合、システムユーザーにはホームディレクトリがありません。コンポーネントを実行するユーザーやグループを設定する方法の詳細については、「コンポーネントを実行するユーザーを設定する」を参照してください。

> **Note**
>
> AWS IoT Greengrass はコンポーネントのセマンティックバージョンを使用します。セマンティックバージョンは、major.minor.patch といった番号システムに準拠します。例えば、バージョン `1.0.0` は、コンポーネントの最初のメジャーリリースを表しています。詳細については、「セマンティックバージョンの仕様」を参照してください。

トピック

- コンポーネントライフサイクル

- コンポーネントタイプ
- AWS IoT Greengrass コンポーネントの作成
- ローカルデプロイで AWS IoT Greengrass コンポーネントをテストする
- コアデバイスにデプロイするコンポーネントをパブリッシュ
- AWS サービスとやり取り
- Docker コンテナの実行
- AWS IoT Greengrass コンポーネントレシピのリファレンス
- コンポーネントの環境変数リファレンス

コンポーネントライフサイクル

コンポーネントライフサイクルは、AWS IoT Greengrass Core ソフトウェアがコンポーネントをインストールして実行するために使用するステージを定義します。各ステージでは、スクリプトと、コンポーネントの動作を指定するその他の情報を定義します。例えば、コンポーネントをインストールするとき、AWS IoT Greengrass Core ソフトウェアはそのコンポーネントの `Install` ライフサイクルスクリプトを実行します。コアデバイス上のコンポーネントには、次のライフサイクルステータスがあります。

- `NEW` - コンポーネントのレシピとアーティファクトはコアデバイスに読み込まれていますが、コンポーネントはインストールされていません。コンポーネントがこの状態になった後、コンポーネントは install script を実行します。
- `INSTALLED` - コンポーネントがコアデバイスにインストールされています。コンポーネントは、install script を実行した後にこの状態に入ります。
- `STARTING` - コアデバイス上でコンポーネントが開始されています。コンポーネントは、startup script を実行したときにこの状態に入ります。起動に成功すると、コンポーネントは `RUNNING` 状態に入ります。
- `RUNNING` - コアデバイス上でコンポーネントが実行されています。コンポーネントは、run script を実行したとき、またはスタートアップスクリプトからのアクティブなバックグラウンドプロセスがある場合に、この状態になります。
- `FINISHED` - コンポーネントが正常に実行され、実行が完了しました。
- `STOPPING` - コンポーネントは停止しています。コンポーネントは、shutdown script を実行したときにこの状態に入ります。
- `ERRORED` - コンポーネントでエラーが発生しました。コンポーネントがこの状態に入ると、recover script を実行します。その後、コンポーネントは再起動して、通常の使用に戻ろうと

試みます。コンポーネントが正常に実行されずに 3 回 ERRORED 状態になると、コンポーネントは BROKEN になります。

- BROKEN - コンポーネントでエラーが複数回発生し、回復できません。修復するには、コンポーネントをもう一度デプロイする必要があります。

コンポーネントタイプ

コンポーネントタイプは、AWS IoT Greengrass Core ソフトウェアがコンポーネントを実行する方法を指定します。コンポーネントには次のタイプがあります。

- Nucleus (`aws.greengrass.nucleus`)

  Greengrass nucleus は、AWS IoT Greengrass Core ソフトウェアの最低限の機能のみを提供するコンポーネントです。詳細については、「Greengrass nucleus」を参照してください。

- プラグイン (`aws.greengrass.plugin`)

  Greengrass nucleus は nucleus と同じ Java 仮想マシン (JVM) で、プラグインコンポーネントを実行します。コアデバイス上のプラグインコンポーネントのバージョンが変更されると、nucleus は再起動します。プラグインコンポーネントをインストールして実行するには、Greengrass nucleus をシステムサービスとして実行するように設定する必要があります。詳細については、「Greengrass nucleus をシステムサービスとして設定する」を参照してください。

  AWS から提供されるコンポーネントにはプラグインコンポーネントが複数あり、Greengrass nucleus と直接インターフェイス接続することができます。プラグインコンポーネントは Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

- ジェネリック (`aws.greengrass.generic`)

  Greengrass nucleus は、コンポーネントがライフサイクルを定義している場合、ジェネリックコンポーネントのライフサイクルスクリプトを実行します。

  このタイプは、カスタムコンポーネントのデフォルトタイプです。

- Lambda (`aws.greengrass.lambda`)

  Greengrass nucleus は、Lambda ランチャーコンポーネントを使用して Lambda 関数コンポーネントを実行します。

Lambda 関数からコンポーネントを作成すると、コンポーネントはこのタイプになります。詳細については、「AWS Lambda 関数を実行する」を参照してください。

> **Note**
>
> レシピでコンポーネントタイプを指定することは推奨しません。AWS IoT Greengrass がコンポーネントを作成するときにタイプを設定します。

AWS IoT Greengrass コンポーネントの作成

ローカル開発コンピュータまたは Greengrass コアデバイスでカスタム AWS IoT Greengrass コンポーネントを開発できます。 AWS IoT Greengrass には、事前定義されたコンポーネントテンプレートとコミュニティコンポーネントからコンポーネントを作成、構築、公開するのに役立つ AWS IoT Greengrass Development Kit コマンドラインインターフェイス (GDK CLI) が用意されています。組み込みのシェルコマンドを実行して、コンポーネントを作成、ビルド、パブリッシュすることもできます。カスタム Greengrass コンポーネントを作成するには、次のオプションから選択します。

- Greengrass Development Kit CLI を使用する

  GDK CLI を使用して、ローカル開発コンピュータ上でコンポーネントを開発します。GDK CLI は、コンポーネントのソースコードを recipe とアーティファクトにビルドしてパッケージ化し、プライベートコンポーネントとして AWS IoT Greengrass サービスにパブリッシュできます。コンポーネントをパブリッシュするときにコンポーネントのバージョンとアーティファクト URI を自動的に更新するように GDK CLI を設定できるため、recipe を毎回更新する必要はありません。GDK CLI を使用してコンポーネントを開発する場合、テンプレートまたは Greengrass ソフトウェアカタログにあるコミュニティコンポーネントを使って開始することができます。詳細については、「AWS IoT Greengrass Development Kit Command-Line Interface」を参照してください。

- 組み込みのシェルコマンドを実行する

  組み込みのシェルコマンドを実行して、ローカル開発コンピュータまたは Greengrass コアデバイス上でコンポーネントを開発できます。シェルコマンドを使用して、コンポーネントのソースコードをアーティファクトにコピーするか構築します。コンポーネントの新しいバージョンを作成するたびに、新しいコンポーネントバージョンで recipe を作成または更新する必要があります。コ

ンポーネントを AWS IoT Greengrass サービスにパブリッシュするときには、recipe 内の各コンポーネントアーティファクトに URI を更新する必要があります。

トピック

コンポーネントを作成する (GDK CLI)

このセクションの指示に従って、GDK CLI を使用してコンポーネントを作成および構築します。

Greengrass コンポーネントを開発するには (GDK CLI)

1. GDK CLI をまだインストールしていない場合は、開発コンピュータにインストールします。詳細については、「AWS IoT Greengrass Development Kit Command-Line Interface をインストールまたは更新する」を参照してください。
2. コンポーネントフォルダを作成するフォルダに移動します。

   Linux or Unix

   ```
   mkdir ~/greengrassv2
   cd ~/greengrassv2
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir %USERPROFILE%\greengrassv2
   cd %USERPROFILE%\greengrassv2
   ```

   PowerShell

   ```
   mkdir ~/greengrassv2
   cd ~/greengrassv2
   ```

3. ダウンロードするコンポーネントテンプレートまたはコミュニティコンポーネントを選択します。GDK CLI がテンプレートまたはコミュニティコンポーネントをダウンロードし、実用的な例を使って開始することができます。component list コマンドを使用して、利用可能なテンプレートとコミュニティコンポーネントのリストを取得します。

- コンポーネントテンプレートをリスト表示するには、次のコマンドを実行します。レスポンスの各行には、テンプレートの名前とプログラミング言語が含まれています。

```
gdk component list --template
```

- コミュニティコンポーネントをリスト表示するには、次のコマンドを実行します。

```
gdk component list --repository
```

4. GDK CLI がテンプレートまたはコミュニティコンポーネントをダウンロードするコンポーネントフォルダを作成および変更します。をコンポーネントの名前、またはこのコンポーネントフォルダを識別するのに役立つ別の名前*HelloWorld*に置き換えます。

   Linux or Unix

```
mkdir HelloWorld
cd HelloWorld
```

   Windows Command Prompt (CMD)

```
mkdir HelloWorld
cd HelloWorld
```

   PowerShell

```
mkdir HelloWorld
cd HelloWorld
```

5. テンプレートまたはコミュニティコンポーネントを現在のフォルダにダウンロードします。component init コマンドを使用します。

   - テンプレートからコンポーネントフォルダを作成するには、次のコマンドを実行します。をテンプレートの名前*HelloWorld*に置き換え、*Python* をプログラミング言語の名前に置き換えます。

```
gdk component init --template HelloWorld --language python
```

   - コミュニティコンポーネントからコンポーネントフォルダを作成するには、次のコマンドを実行します。をコミュニティコンポーネントの名前*ComponentName*に置き換えます。

```
gdk component init --repository ComponentName
```

> **Note**
>
> GDK CLI v1.0.0 を使用している場合は、このコマンドは空のフォルダで実行する必要があります。GDK CLI がテンプレートまたはコミュニティコンポーネントを現在のフォルダにダウンロードします。
> GDK CLI v1.1.0 以降を使用する場合、`--name` 引数を使用して、GDK CLI がテンプレートまたはコミュニティコンポーネントをダウンロードするフォルダを指定することができます。この引数を使用する場合は、存在しないフォルダを指定します。GDK CLI によってフォルダが作成されます。この引数を指定しなかった場合、GDK CLI は現在のフォルダを使用しますが、このフォルダは空である必要があります。

6. GDK CLI は `gdk-config.json` という名の GDK CLI 設定ファイルから読み込むことで、コンポーネントをビルドおよびパブリッシュします。この設定ファイルは、コンポーネントフォルダのルートにあります。前の手順では、このファイルが作成されました。このステップでは、`gdk-config.json` をコンポーネントに関する情報で更新します。以下の操作を実行します。

   a. テキストエディタで `gdk-config.json` を開きます。

   b. (オプション) コンポーネントの名前を変更します。コンポーネント名は、`component` オブジェクトのキーです。

   c. コンポーネントの作成者を変更します。

   d. (オプション) コンポーネントのバージョンを変更します。次のいずれかを指定します。

      - `NEXT_PATCH` - このオプションを選択すると、コンポーネントをパブリッシュするときに GDK CLI がバージョンを設定します。GDK CLI は AWS IoT Greengrass サービスをクエリして、コンポーネントの最新の公開バージョンを特定します。次に、そのバージョンの後の次のパッチバージョンにバージョンを設定します。コンポーネントをまだパブリッシュしていない場合は、GDK CLI はバージョン`1.0.0` を使用します。

        このオプションを選択した場合、Greengrass CLI を使用して、 AWS IoT Greengrass Core ソフトウェアを実行するローカル開発コンピュータにコンポーネントをローカルにデプロイしてテストすることはできません。ローカルデプロイを有効にするには、代わりにセマンティックバージョンを指定する必要があります。

- **`1.0.0`** などのセマンティックバージョンです。セマンティックバージョンは、major.minor.patch という番号方式になっています。詳細については、「セマンティックバージョンの仕様」を参照してください。

  コンポーネントをデプロイしてテストする Greengrass コアデバイスでコンポーネントを開発する場合は、このオプションを選択します。Greengrass CLI を使用してローカルデプロイを作成する場合は、特定のバージョンでコンポーネントを構築する必要があります。

e. (オプション) コンポーネントのビルド設定を変更します。ビルド設定では、GDK CLI がコンポーネントのソースをアーティファクトにビルドする方法を定義します。次の `build_system` のオプションの中から選択します。

- `zip` - コンポーネントのフォルダを ZIP ファイルにパッケージ化し、コンポーネントの唯一のアーティファクトとして定義します。次のタイプのコンポーネントには、このオプションを選択します。
  - Python や など、解釈されたプログラミング言語を使用するコンポーネント JavaScript。
  - 機械学習モデルやその他のリソースなどの、コード以外のファイルをパッケージ化するコンポーネント。

  GDK CLI は、コンポーネントのフォルダをコンポーネントフォルダと同じ名前の zip ファイルに圧縮します。例えば、コンポーネントフォルダの名前が `HelloWorld` の場合、GDK CLI は `HelloWorld.zip` という名前の zip ファイルを作成します。

  > **Note**
  >
  > Windows デバイスで GDK CLI バージョン 1.0.0 を使用する場合、コンポーネントフォルダと zip ファイル名には小文字のみを含める必要があります。

  GDK CLI がコンポーネントのフォルダを zip ファイルに圧縮する際、次のファイルはスキップされます。
  - `gdk-config.json` ファイル
  - recipe ファイル (`recipe.json` または `recipe.yaml`)
  - ビルドフォルダ (`greengrass-build` など)

- `maven` - `mvn clean package` コマンドを実行して、コンポーネントのソースをアーティファクト内にビルドします。Java コンポーネントなどの Maven を使用するコンポーネントの場合は、このオプションを選択します。

  Windows デバイスでは、この機能は GDK CLI v1.1.0 以降で利用できます。

- `gradle` - `gradle build` コマンドを実行して、コンポーネントのソースをアーティファクト内にビルドします。Gradle を使用するコンポーネントの場合は、このオプションを選択します。この機能は GDK CLI v1.1.0 以降で利用できます。

  `gradle` ビルドシステムは、ビルドファイルとして Kotlin DSL をサポートしています。この機能は GDK CLI v1.2.0 以降で利用できます。

- `gradlew` - `gradlew` コマンドを実行して、コンポーネントのソースをアーティファクト内にビルドします。Gradle Wrapper を使用するコンポーネントの場合は、このオプションを選択します。

  この機能は GDK CLI v1.2.0 以降で利用できます。

- `custom` - カスタムコマンドを実行して、コンポーネントのソースを recipe とアーティファクトにビルドします。`custom_build_command` パラメータでカスタムコマンドを指定します。

f. `build_system` の `custom` を指定する場合は、`build` オブジェクトに `custom_build_command` を追加します。`custom_build_command` で、1 つの文字列または文字列のリストを指定します。各文字列が、コマンド内の単語になります。例えば、C++ コンポーネントのカスタムビルドコマンドを実行する場合は、**`["cmake", "--build", "build", "--config", "Release"]`** を指定することができます。

g. GDK CLI v1.1.0 以降を使用する場合、`--bucket` 引数を指定して、GDK CLI がコンポーネントのアーティファクトをアップロードする S3 バケットを指定します。この引数を指定しない場合、GDK CLI は名前が の S3 バケットにアップロードします。ここで*`bucket`*-*`region`*-*`accountId`*、*`bucket`* と *`region`* は で指定した値gdk-config.json、*`accountId`* は AWS アカウント ID です。GDK CLI は、バケットが存在しない場合に作成します。

   コンポーネントのパブリッシュ設定を変更します。以下の操作を実行します。

   i. コンポーネントのアーティファクトをホストするために使用する S3 バケット名を指定します。

   ii. GDK CLI AWS リージョン がコンポーネントを発行する を指定します。

このステップを終了した時点で、gdk-config.json ファイルは、次の例のようになります。

```
{
  "component": {
    "com.example.PythonHelloWorld": {
      "author": "Amazon",
      "version": "NEXT_PATCH",
      "build": {
        "build_system" : "zip"
      },
      "publish": {
        "bucket": "greengrass-component-artifacts",
        "region": "us-west-2"
      }
    }
  },
  "gdk_version": "1.0.0"
}
```

7. recipe.yaml または recipe.json という名のコンポーネント recipe ファイルを更新します。以下の操作を実行します。

   a. zip ビルドシステムを使用するテンプレートまたはコミュニティコンポーネントをダウンロードした場合は、ZIP アーティファクト名がコンポーネントフォルダの名前と一致していることを確認してください。GDK CLI は、コンポーネントのフォルダをコンポーネントフォルダと同じ名前の zip ファイルに圧縮します。recipe では、コンポーネントアーティファクトのリストと、ZIP アーティファクト内のファイルを使用するライフサイクルスクリプトの中に、ZIP アーティファクトの名前が含まれます。ZIP ファイル名がコンポーネントフォルダの名前と一致するように Artifacts および Lifecycle の定義を更新します。次の recipe 例では、Artifacts と Lifecycle の定義内の zip ファイルの名前が強調表示されています。

      JSON

      ```
      {
        ...
        "Manifests": [
          {
            "Platform": {
              "os": "all"
      ```

```
      },
      "Artifacts": [
        {
          "URI": "s3://{COMPONENT_NAME}/{COMPONENT_VERSION}/HelloWorld.zip",
          "Unarchive": "ZIP"
        }
      ],
      "Lifecycle": {
        "run": "python3 -u {artifacts:decompressedPath}/HelloWorld/main.py
 {configuration:/Message}"
      }
    }
  ]
}
```

YAML

```
---
...
Manifests:
  - Platform:
      os: all
    Artifacts:
      - URI: "s3://BUCKET_NAME/COMPONENT_NAME/
COMPONENT_VERSION/HelloWorld.zip"
        Unarchive: ZIP
    Lifecycle:
      run: "python3 -u {artifacts:decompressedPath}/HelloWorld/main.py
 {configuration:/Message}"
```

b. (オプション) コンポーネントの説明、デフォルト設定、アーティファクト、ライフサイクルスクリプト、およびプラットフォームサポートを更新します。詳細については、「AWS IoT Greengrass コンポーネントレシピのリファレンス」を参照してください。

このステップを終了した時点で、recipe ファイルは、次の例のようになります。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "{COMPONENT_NAME}",
  "ComponentVersion": "{COMPONENT_VERSION}",
```

```
  "ComponentDescription": "This is a simple Hello World component written in
 Python.",
  "ComponentPublisher": "{COMPONENT_AUTHOR}",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "World"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "all"
      },
      "Artifacts": [
        {
          "URI": "s3://{COMPONENT_NAME}/{COMPONENT_VERSION}/HelloWorld.zip",
          "Unarchive": "ZIP"
        }
      ],
      "Lifecycle": {
        "run": "python3 -u {artifacts:decompressedPath}/HelloWorld/main.py
 {configuration:/Message}"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: "2020-01-25"
ComponentName: "{COMPONENT_NAME}"
ComponentVersion: "{COMPONENT_VERSION}"
ComponentDescription: "This is a simple Hello World component written in
 Python."
ComponentPublisher: "{COMPONENT_AUTHOR}"
ComponentConfiguration:
  DefaultConfiguration:
    Message: "World"
Manifests:
  - Platform:
      os: all
    Artifacts:
```

```
      - URI: "s3://BUCKET_NAME/COMPONENT_NAME/COMPONENT_VERSION/HelloWorld.zip"
        Unarchive: ZIP
    Lifecycle:
      run: "python3 -u {artifacts:decompressedPath}/HelloWorld/main.py
{configuration:/Message}"
```

8. Greengrass コンポーネントを開発して構築します。コンポーネントビルドコマンドは、コンポーネントフォルダにある `greengrass-build` フォルダに recipe とアーティファクトを作成します。以下のコマンドを実行します。

   ```
   gdk component build
   ```

コンポーネントをテストする準備ができたら、GDK CLI を使用して、 AWS IoT Greengrass サービスにパブリッシュします。その後に、コンポーネントを Greengrass コアデバイスにデプロイできます。詳細については、「コアデバイスにデプロイするコンポーネントをパブリッシュ」を参照してください。

コンポーネントを作成する (シェルコマンド)

このセクションの手順に従って、複数のコンポーネントのソースコードとアーティファクトが含まれる recipe とアーティファクトフォルダを作成します。

Greengrass コンポーネントを開発するには (シェルコマンド)

1. recipe とアーティファクトのサブフォルダを含むコンポーネントのフォルダを作成します。Greengrass コアデバイスで次のコマンドを実行してこれらのフォルダを作成し、コンポーネントフォルダに変更します。*~/greengrassv2* または *%USERPROFILE%\greengrassv2* をローカル開発に使用するフォルダへのパスに置き換えます。

   Linux or Unix

   ```
   mkdir -p ~/greengrassv2/{recipes,artifacts}
   cd ~/greengrassv2
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir %USERPROFILE%\greengrassv2\\recipes, %USERPROFILE%\greengrassv2\\artifacts
   cd %USERPROFILE%\greengrassv2
   ```

PowerShell

```
mkdir ~/greengrassv2/recipes, ~/greengrassv2/artifacts
cd ~/greengrassv2
```

2. テキストエディタを使用して、コンポーネントのメタデータ、パラメータ、依存関係、ライフサイクル、プラットフォーム機能を定義する recipe ファイルを作成します。recipe ファイル名にコンポーネントのバージョンを含めるようにして、どの recipe がどのコンポーネントバージョンを反映しているのかを特定できるようにします。recipe には YAML 形式または JSON 形式を選択できます。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを作成できます。

   JSON

   ```
   nano recipes/com.example.HelloWorld-1.0.0.json
   ```

   YAML

   ```
   nano recipes/com.example.HelloWorld-1.0.0.yaml
   ```

   > **Note**
   >
   > AWS IoT Greengrass はコンポーネントのセマンティックバージョンを使用します。セマンティックバージョンは、major.minor.patch といった番号システムに準拠します。例えば、バージョン `1.0.0` は、コンポーネントの最初のメジャーリリースを表しています。詳細については、「セマンティックバージョンの仕様」を参照してください。

3. コンポーネントの recipe を定義します。詳細については、「AWS IoT Greengrass コンポーネントレシピのリファレンス」を参照してください。

   recipe は、次 Hello World の recipe 例のようになります。

   JSON

   ```
   {
     "RecipeFormatVersion": "2020-01-25",
   ```

```
  "ComponentName": "com.example.HelloWorld",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "My first AWS IoT Greengrass component.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "world"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "run": "python3 -u {artifacts:path}/hello_world.py {configuration:/
Message}"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "run": "py -3 -u {artifacts:path}/hello_world.py {configuration:/
Message}"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: My first AWS IoT Greengrass component.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    Message: world
Manifests:
```

```
  - Platform:
      os: linux
    Lifecycle:
      run: |
        python3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
  - Platform:
      os: windows
    Lifecycle:
      run: |
        py -3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
```

この recipe は、Hello World Python スクリプトを実行します。次のスクリプト例のようになります。

```
import sys

message = "Hello, %s!" % sys.argv[1]

# Print the message to stdout, which Greengrass saves in a log file.
print(message)
```

4. 開発するコンポーネントバージョン用のフォルダを作成します。どのアーティファクトがどのコンポーネントバージョンのものなのかを識別できるように、各コンポーネントバージョンのアーティファクトに対して個別のフォルダを使用することをお勧めします。以下のコマンドを実行します。

   Linux or Unix

   ```
   mkdir -p artifacts/com.example.HelloWorld/1.0.0
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir artifacts/com.example.HelloWorld/1.0.0
   ```

   PowerShell

   ```
   mkdir artifacts/com.example.HelloWorld/1.0.0
   ```

> **Important**
>
> アーティファクトフォルダのパスには、次のフォーマットを使用する必要があります。recipe で指定したコンポーネント名とバージョンを含めてください。
>
> ```
> artifacts/componentName/componentVersion/
> ```

5. 前のステップで作成したフォルダに、コンポーネントのアーティファクトを作成します。アーティファクトには、ソフトウェア、イメージ、およびその他のコンポーネントが使用するバイナリを含めることができます。

   コンポーネントの準備ができたら、コンポーネントをテストします。

ローカルデプロイで AWS IoT Greengrass コンポーネントをテストする

コアデバイスで Greengrass コンポーネントを開発する場合は、ローカルデプロイを作成してインストールしてテストできます。このセクションのステップに従ってローカルデプロイを作成します。

ローカルデプロイコンピュータなど、別のコンピュータでコンポーネントを開発する場合、ローカルデプロイを作成することはできません。代わりに、Greengrass コアデバイスにデプロイしてテストできるように、コンポーネントを AWS IoT Greengrass サービスに公開します。詳細については、「コアデバイスにデプロイするコンポーネントをパブリッシュ と デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

Greengrass コアデバイスでコンポーネントをテストするには

1. コアデバイスは、コンポーネントの更新などのイベントをログに記録します。このログファイルを表示して、無効なレシピなど、コンポーネントのエラーを検出してトラブルシューティングできます。このログファイルには、コンポーネントが標準出力 (stdout) に出力するメッセージも表示されます。コアデバイスで追加のターミナルセッションを開き、新しいログメッセージをリアルタイムで監視することをお勧めします。SSH などの新しいターミナルセッションを開き、次のコマンドを実行してログを表示します。をAWS IoT Greengrassルートフォルダへのパス/*greengrass/v2*に置き換えます。

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

PowerShell

```
gc C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
```

コンポーネントのログファイルを表示することもできます。

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
```

PowerShell

```
gc C:\greengrass\v2\logs\com.example.HelloWorld.log -Tail 10 -Wait
```

2. 元のターミナルセッションで、次のコマンドを実行して、コアデバイスをコンポーネントで更新します。をAWS IoT Greengrassルートフォルダへのパス*/greengrass/v2*に置き換え、*~/greengrassv2* をローカル開発フォルダへのパスに置き換えます。

Linux or Unix

```
sudo /greengrass/v2/bin/greengrass-cli deployment create \
  --recipeDir ~/greengrassv2/recipes \
  --artifactDir ~/greengrassv2/artifacts \
  --merge "com.example.HelloWorld=1.0.0"
```

Windows Command Prompt (CMD)

```
C:\greengrass\v2\bin\greengrass-cli deployment create ^
  --recipeDir %USERPROFILE%\greengrassv2\recipes ^
  --artifactDir %USERPROFILE%\greengrassv2\artifacts ^
  --merge "com.example.HelloWorld=1.0.0"
```

PowerShell

```
C:\greengrass\v2\bin\greengrass-cli deployment create `
  --recipeDir ~/greengrassv2/recipes `
  --artifactDir ~/greengrassv2/artifacts `
  --merge "com.example.HelloWorld=1.0.0"
```

> **Note**
>
> また、greengrass-cli deployment create コマンドを使用して、コンポーネントの設定パラメータの値を設定することもできます。詳細については、「作成」を参照してください。

3. greengrass-cli deployment status コマンドを使用して、コンポーネントのデプロイの進行状況を監視します。

   Unix or Linux

   ```
   sudo /greengrass/v2/bin/greengrass-cli deployment status \
     -i deployment-id
   ```

   Windows Command Prompt (CMD)

   ```
   C:\greengrass\v2\bin\greengrass-cli deployment status ^
     -i deployment-id
   ```

   PowerShell

   ```
   C:\greengrass\v2\bin\greengrass-cli deployment status `
     -i deployment-id
   ```

4. Greengrass コアデバイスで実行されるコンポーネントをテストします。このバージョンのコンポーネントを終了したら、AWS IoT Greengrass サービスにアップロードできます。その後、コンポーネントを他のコアデバイスに展開できます。詳細については、「コアデバイスにデプロイするコンポーネントをパブリッシュ」を参照してください。

コアデバイスにデプロイするコンポーネントをパブリッシュ

コンポーネントのバージョンを構築または完成した後、AWS IoT Greengrass サービスにパブリッシュできます。次に、Greengrass コアデバイスにデプロイできます。

Greengrass Development Kit CLI (GDK CLI) を使用してコンポーネントを開発と構築する場合、GDK CLI を使用してコンポーネントを AWS クラウド にパブリッシュできます。それ以外の場合、組込み型のシェルコマンドと AWS CLI を使用してコンポーネントをパブリッシュします。

AWS CloudFormation を使用して、テンプレートでコンポーネントと他の AWS リソースを作成することもできます。詳細については、「AWS CloudFormation ユーザーガイド」の「AWS CloudFormation とは」と「AWS::GreengrassV2::ComponentVersion」を参照してください。

トピック

- コンポーネントをパブリッシュ (GDK CLI)
- コンポーネントをパブリッシュ (シェルコマンド)

コンポーネントをパブリッシュ (GDK CLI)

このセクションの指示に従って、GDK CLI を使用してコンポーネントをパブリッシュします。GDK CLI は、S3 バケットにビルドアーティファクトをアップロード、recipe のアーティファクト URI を更新、recipe からコンポーネントを作成します。S3 バケットとリージョンが GDK CLI 設定ファイルを使用するように指定します。

GDK CLI v1.1.0 以降を使用する場合、`--bucket` 引数を指定して、GDK CLI がコンポーネントのアーティファクトをアップロードする S3 バケットを指定します。この引数を指定しない場合、GDK CLI は名前が *`bucket`*`-`*`region`*`-`*`accountId`* である S3 バケットにアップロードします。ここでは、`gdk-config.json` で指定する値は *`bucket`* と *`region`* であり、*`accountId`* は AWS アカウント ID です。GDK CLI は、バケットが存在しない場合に作成します。

> **Important**
>
> コアデバイスのロールは、デフォルトで S3 バケットへのアクセスを許可しません。この S3 バケットを初めて使用する場合、コアデバイスがこの S3 バケットからコンポーネントアーティファクトを取得するため、ロールに許可を追加する必要があります。詳細については、「コンポーネントのアーティファクトの S3 バケットへのアクセスを許可する」を参照してください。

Greengrass コンポーネントをパブリッシュするには (GDK CLI)

1. コマンドプロントまたはターミナルでコンポーネントフォルダを開きます。
2. まだ作成していない場合、Greengrass コンポーネントを作成します。コンポーネントビルドコマンドは、コンポーネントフォルダにある `greengrass-build` フォルダに recipe とアーティファクトを作成します。以下のコマンドを実行します。

   ```
   gdk component build
   ```

3. コンポーネントを AWS クラウド にパブリッシュします。コンポーネントをパブリッシュコマンドは、コンポーネントのアーティファクトを Amazon S3 にアップロードし、各アーティファクトの URI でコンポーネントの recipe を更新します。次に、コンポーネントを AWS IoT Greengrass サービスに作成します。

   > **Note**
   >
   > AWS IoT Greengrass は、コンポーネントの作成時に各アーティファクトのダイジェストを計算します。つまり、コンポーネントを作成した後、S3 バケットのアーティファクトファイルを修正することはできません。そうした場合、ファイルダイジェストが一致しないため、このコンポーネントを含むデプロイは失敗します。アーティファクトファイルを修正する場合、コンポーネントの新しいバージョンを作成する必要があります。

   GDK CLI 設定ファイルのコンポーネントバージョンに `NEXT_PATCH` を指定した場合、GDK CLI は AWS IoT Greengrass サービスにまだ存在しない次のパッチバージョンを使用します。

   以下のコマンドを実行します。

   ```
   gdk component publish
   ```

   出力には、GDK CLI が作成したコンポーネントのバージョンが示されます。

   コンポーネントをパブリッシュした後、コンポーネントをコアデバイスにデプロイできます。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

コンポーネントをパブリッシュ (シェルコマンド)

シェルコマンドと AWS Command Line Interface (AWS CLI) を使用してコンポーネントをパブリッシュするには、次の手順を実行します。コンポーネントをパブリッシュするときは、次の手順を実行します。

1. S3 バケットにコンポーネントアーティファクトをパブリッシュします。
2. 各アーティファクトの Amazon S3 URI をコンポーネント recipe に追加します。
3. コンポーネント recipe から AWS IoT Greengrass でコンポーネントバージョンを作成します。

> **Note**
>
> アップロードする各コンポーネントバージョンは一意である必要があります。アップロード後は編集できないため、必ず正しいコンポーネントバージョンをアップロードしてください。

これらの手順に従って、開発コンピュータまたは Greengrass コアデバイスからコンポーネントをパブリッシュできます。

コンポーネントをパブリッシュするには (シェルコマンド)

1. コンポーネントが、AWS IoT Greengrass サービスに存在するバージョンを使用している場合、コンポーネントのバージョンを変更する必要があります。テキストエディタで recipe を開き、バージョンをインクリメントしてファイルを保存します。コンポーネントに加えた変更を反映する新しいバージョンを選択します。

   > **Note**
   >
   > AWS IoT Greengrass はコンポーネントのセマンティックバージョンを使用します。セマンティックバージョンは、major.minor.patch といった番号システムに準拠します。例えば、バージョン `1.0.0` は、コンポーネントの最初のメジャーリリースを表しています。詳細については、「セマンティックバージョンの仕様」を参照してください。

2. コンポーネントにアーティファクトがある場合、次の手順を実行します。

   a. AWS アカウント の S3 バケットにコンポーネントのアーティファクトをパブリッシュします。

> **Tip**
>
> S3 バケットのアーティファクトへのパスに、コンポーネント名とバージョンを含めることをお勧めします。この命名規則は、以前のバージョンのコンポーネントが使用していたアーティファクトを維持するうえで役立ち、以前のコンポーネントバージョンを引き続きサポートできるようにします。

次のコマンドを実行して、アーティファクトファイルを S3 バケットにパブリッシュします。*DOC-EXAMPLE-BUCKET* をバケットの名前に置き換え、*ifacts/com.example.HelloWorld/#################### 1.0.0/artifact.py*。

```
aws s3 cp artifacts/com.example.HelloWorld/1.0.0/artifact.py s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/artifact.py
```

> **Important**
>
> コアデバイスのロールは、デフォルトで S3 バケットへのアクセスを許可しません。この S3 バケットを初めて使用する場合、コアデバイスがこの S3 バケットからコンポーネントアーティファクトを取得するため、ロールに許可を追加する必要があります。詳細については、「コンポーネントのアーティファクトの S3 バケットへのアクセスを許可する」を参照してください。

b. `Artifacts` という名前のリストが存在しない場合、コンポーネント recipe に追加します。`Artifacts` リストは各マニフェストに表示され、サポートする各プラットフォームにおけるコンポーネントの要件 (またはすべてのプラットフォームに対するコンポーネントのデフォルト要件) を定義します。

c. 各アーティファクトをアーティファクトのリストに追加、あるいは既存アーティファクトの URI を更新します。Amazon S3 URI は、バケット名とバケット内のアーティファクトオブジェクトへのパスで設定されます。アーティファクトの Amazon S3 URI は次の例のようになります。

```
s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/artifact.py
```

これらのステップが完了したら、recipe の `Artifacts` リストは次のようになります。

JSON

```
{
  ...
  "Manifests": [
    {
      "Lifecycle": {
        ...
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/MyGreengrassComponent/1.0.0/artifact.py",
          "Unarchive": "NONE"
        }
      ]
    }
  ]
}
```

> **Note**
>
> ZIP アーティファクトに "Unarchive": "ZIP" オプションを追加して AWS IoT Greengrass Core ソフトウェアを設定して、コンポーネントのデプロイ時にアーティファクトを解凍できます。

YAML

```
...
Manifests:
  - Lifecycle:
      ...
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/MyGreengrassComponent/1.0.0/artifact.py
        Unarchive: NONE
```

> **Note**
>
> コンポーネントのデプロイ時に ZIP アーティファクトを解凍するため、`Unarchive: ZIP` オプションを使用して AWS IoT Greengrass Core ソフトウェアを設定できます。コンポーネントで ZIP アーティファクトを使用する方法の詳細については、アーティファクト: decompressedPath recipe 変数を参照してください。

recipe の詳細については、「AWS IoT Greengrass コンポーネントレシピのリファレンス」を参照してください。

3. AWS IoT Greengrass コンソールを使用して、recipe ファイルからコンポーネントを作成します。

   次のコマンドを実行して、recipe ファイルからコンポーネントを作成します。このコマンドは、コンポーネントを作成して AWS アカウント のプライベート AWS IoT Greengrass コンポーネントとしてパブリッシュします。*`path/to/recipeFile`* を recipe ファイルへのパスに置き換えます。

   ```
   aws greengrassv2 create-component-version --inline-recipe fileb://path/to/
   recipeFile
   ```

   レスポンスから `arn` をコピーして、次のステップでコンポーネントの状態をチェックします。

   > **Note**
   >
   > AWS IoT Greengrass は、コンポーネントの作成時に各アーティファクトのダイジェストを計算します。つまり、コンポーネントを作成した後、S3 バケットのアーティファクトファイルを修正することはできません。そうした場合、ファイルダイジェストが一致しないため、このコンポーネントを含むデプロイは失敗します。アーティファクトファイルを修正する場合、コンポーネントの新しいバージョンを作成する必要があります。

4. AWS IoT Greengrass サービスの各コンポーネントには状態があります。次のコマンドを実行して、この手順でパブリッシュするコンポーネントバージョンの状態を確認します。*`com.example.HelloWorld`* および *`1.0.0`* をクエリするコンポーネントバージョンに置き換えます。`arn` を、前のステップで書き留めた ARN に置き換えます。

```
aws greengrassv2 describe-component --arn "arn:aws:greengrass:region:account-
id:components:com.example.HelloWorld:versions:1.0.0"
```

このオペレーションは、コンポーネントのメタデータを含むレスポンスを返します。メタデータには、コンポーネントの状態とエラーを含む `status` オブジェクトが含まれています (該当する場合)。

コンポーネントの状態が `DEPLOYABLE` な場合、コンポーネントをデバイスにデプロイできます。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

AWS サービスとやり取り

Greengrass コアデバイスは、TLS 相互認証プロトコルを使用して AWS IoT Core に接続するために、X.509 証明書を使用します。これらの証明書は、AWS 認証情報 (通常はアクセスキー ID とシークレットアクセスキー) がなくても、デバイスが AWS IoT とやり取りできるようにします。その他の AWS サービスは、サービスエンドポイントで API オペレーションを呼び出す際に、X.509 証明書ではなく AWS 認証情報が必要になります。AWS IoT Core には認証情報プロバイダがあり、これはデバイスが AWS リクエストを認証するために X.509 証明書を使用することを可能にします。AWS IoT 認証情報プロバイダは、X.509 証明書を使用してデバイスを認証し、一時的で制限された権限のセキュリティトークンの形で AWS 認証情報を発行します。デバイスはこのトークンを使用して、すべての AWS リクエストに署名と認証を行うことができます。これにより、Greengrass コアデバイスでは AWS 認証情報を保存する必要がなくなります。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS サービスへの直接呼び出しを認証する」を参照してください。

AWS IoT から認証情報を取得するために、Greengrass コアデバイスは IAM ロールを指す AWS IoT ロールエイリアスを使用します。この IAM ロールは「トークン交換ロール」と呼ばれます。ロールエイリアスとトークン交換ロールは、AWS IoT Greengrass Core ソフトウェアをインストールする際に作成します。コアデバイスが使用するロールエイリアスを指定するには、Greengrass nucleus の `iotRoleAlias` パラメータを設定します。

AWS IoT 認証情報プロバイダは、ユーザーに代わって、コアデバイスに AWS 認証情報を提供するためのトークン交換ロールを引き受けます。適切な IAM ポリシーをこのロールにアタッチすることで、S3 バケット内のコンポーネントアーティファクトなどの AWS リソースにコアデバイスがアクセスできるようになります。トークン交換ロールを設定する方法の詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

Greengrass コアデバイスは AWS 認証情報をメモリに保存しますが、デフォルトでは、認証情報は 1 時間で期限が切れます。AWS IoT Greengrass Core ソフトウェアが再起動した場合は、認証情報を再度取得する必要があります。UpdateRoleAlias オペレーションを使用して、認証情報が有効である期間を設定できます。

AWS IoT Greengrass はパブリックコンポーネントの 1 つであるトークン交換サービスコンポーネントを提供します。これはカスタムコンポーネントの依存関係として定義でき、AWS サービスとやり取りを可能にします。トークン交換サービスはコンポーネントに環境変数 `AWS_CONTAINER_CREDENTIALS_FULL_URI` を提供します。これは AWS 認証情報を提供するローカルサーバーへの URI を定義するものです。AWS SDK クライアントを作成すると、クライアントはこの環境変数をチェックしてローカルサーバーに接続し、AWS 認証情報を取得して、これを使用して API リクエストに署名します。これにより、コンポーネントで AWS サービスを呼び出すために、AWS SDK などのツールを使用できるようになります。詳細については、「トークン交換サービス」を参照してください。

> **Important**
>
> この方法で AWS 認証情報を取得するサポートは、2016 年 7 月 13 日に AWS の SDK に追加されました。コンポーネントは、その日以降に作成された AWS SDK バージョンを使用する必要があります。詳細については、「Amazon Elastic Container Service デベロッパーガイド」の「サポートされる AWS SDK の使用」を参照してください。

カスタムコンポーネントで AWS 認証情報を取得するには、コンポーネントレシピで `aws.greengrass.TokenExchangeService` を依存関係として定義します。次のサンプルレシピは、boto3 をインストールし、Amazon S3 バケットを一覧表示するためにトークン交換サービスからの AWS 認証情報を使用する Python スクリプトを実行するコンポーネントを定義します。

> **Note**
>
> このサンプルコンポーネントを実行するには、デバイスに `s3:ListAllMyBuckets` アクセス許可が必要です。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

JSON

```
{
```

```
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.ListS3Buckets",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that uses the token exchange service to list
 S3 buckets.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.TokenExchangeService": {
      "VersionRequirement": "^2.0.0",
      "DependencyType": "HARD"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "pip3 install --user boto3",
        "run": "python3 -u {artifacts:path}/list_s3_buckets.py"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "pip3 install --user boto3",
        "run": "py -3 -u {artifacts:path}/list_s3_buckets.py"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.ListS3Buckets
ComponentVersion: '1.0.0'
ComponentDescription: A component that uses the token exchange service to list S3
 buckets.
ComponentPublisher: Amazon
```

```
ComponentDependencies:
  aws.greengrass.TokenExchangeService:
    VersionRequirement: '^2.0.0'
    DependencyType: HARD
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install:
        pip3 install --user boto3
      run: |-
        python3 -u {artifacts:path}/list_s3_buckets.py
  - Platform:
      os: windows
    Lifecycle:
      install:
        pip3 install --user boto3
      run: |-
        py -3 -u {artifacts:path}/list_s3_buckets.py
```

このサンプルコンポーネントは、Amazon S3 バケットを一覧表示する、以下の Python スクリプト (`list_s3_buckets.py`) を実行します。

```
import boto3
import os

try:
    print("Creating boto3 S3 client...")
    s3 = boto3.client('s3')
    print("Successfully created boto3 S3 client")
except Exception as e:
    print("Failed to create boto3 s3 client. Error: " + str(e))
    exit(1)

try:
    print("Listing S3 buckets...")
    response = s3.list_buckets()
    for bucket in response['Buckets']:
        print(f'\t{bucket["Name"]}')
    print("Successfully listed S3 buckets")
except Exception as e:
    print("Failed to list S3 buckets. Error: " + str(e))
```

```
    exit(1)
```

Docker コンテナの実行

AWS IoT Greengrass コンポーネントが、次の場所に保存されているイメージの Docker コンテナを実行するように設定できます。

- Amazon Elastic Container Registry (Amazon ECR) のパブリックイメージリポジトリおよびプライベートイメージリポジトリ
- パブリック Docker Hub リポジトリ
- パブリック Docker の信頼レジストリ
- S3 バケット

カスタムコンポーネントで、Docker イメージ URI をアーティファクトとして含めて、イメージを取得してコアデバイスに実行します。Amazon ECR と Docker Hub イメージの場合、Docker アプリケーションマネージャーコンポーネントを使用して、プライベート Amazon ECR リポジトリ用にイメージのダウンロードと認証情報の管理ができます。

トピック

- 要件
- Amazon ECR または Docker Hub のパブリックイメージから Docker コンテナを実行する
- Amazon ECR のプライベートイメージから Docker コンテナを実行
- Amazon S3 のイメージから Docker コンテナの実行
- Docker コンテナコンポーネントでプロセス間通信の使用
- Docker コンテナコンポーネント (Linux) でAWS 認証情報の使用
- Docker コンテナコンポーネント (Linux) でストリームマネージャーの使用

要件

コンポーネントの Docker コンテナを実行するには、次のものが必要です:

- Greengrass コアデバイス。アカウントをお持ちでない場合は、「チュートリアル: AWS IoT Greengrass V2 の開始方法」を参照してください。
- Docker Engine 1.9.1 以降が Greengrass コアにインストールされていいること。バージョン 20.10 は、AWS IoT Greengrass Core ソフトウェアとの動作が確認されている最新バージョ

ンです。Docker コンテナを実行するコンポーネントをデプロイする前に、コアデバイスに直接、Docker をインストールしておく必要があります。

> **Tip**
>
> コンポーネントのインストール時に、Docker Engine をインストールするようにコアデバイスを設定することもできます。例えば、次のインストールスクリプトは、Docker イメージをロードする前に Docker Engine をインストールします。このインストールスクリプトは、Ubuntu など、 Debian ベースの Linux ディストリビューションに動作します。このコマンドで Docker Engine をインストールするようにコンポーネントを設定する場合、ライフサイクルスクリプトに `RequiresPrivilege` を `true` に設定して、インストールと `docker` コマンドを実行する必要があります。詳細については、「AWS IoT Greengrass コンポーネントレシピのリファレンス」を参照してください。
>
> ```
> apt-get install docker-ce docker-ce-cli containerd.io && docker load -i
>  {artifacts:path}/hello-world.tar
> ```

- Docker コンテナコンポーネントを実行するシステムユーザーには、ルート権限または管理者権限が必要です。権限がない場合は、ルート権限または管理者権限を持たないユーザーとして実行されるように Docker を設定する必要があります。
  - Linux デバイスでは、ユーザーを `docker` グループに追加することで、`sudo` のない `docker` コマンドを呼び出せます。
  - Windows デバイスでは、ユーザーを `docker-users` グループ に追加することで、管理者の権限のない `docker` コマンドを呼び出せます。

  Linux or Unix

  Docker コンテナコンポーネントの実行に使用する `ggc_user` または非ルートユーザーを `docker` グループに追加するには、次のコマンドを実行します。

  ```
  sudo usermod -aG docker ggc_user
  ```

  詳細については、「Docker を非ルートユーザーとして管理する」を参照してください。

  Windows Command Prompt (CMD)

  Docker コンテナコンポーネントの実行に使用する `ggc_user` またはユーザーを `docker-users` グループに追加するには、次のコマンドを管理者として実行します。

```
net localgroup docker-users ggc_user /add
```

Windows PowerShell

Docker コンテナコンポーネントの実行に使用する ggc_user またはユーザーを docker-users グループに追加するには、次のコマンドを管理者として実行します。

```
Add-LocalGroupMember -Group docker-users -Member ggc_user
```

- Docker コンテナ内にある Docker コンテナコンポーネントによってアクセスされるボリュームとしてマウントされたファイル。
- AWS IoT Greengrass Core ソフトウェアがネットワークプロキシを使用するように設定している場合、Docker が同じプロキシサーバーを使用するように設定する必要があります。

これらの要件に加えて、環境に該当する場合、次の要件も満たす必要があります:

- Docker Compose を使用して Docker コンテナを作成して起動するには、Greengrass コアデバイスに Docker Compose をインストールして、Docker Compose ファイルを S3 バケットにアップロードします。Compose ファイルは、コンポーネントと同じ AWS アカウント と AWS リージョン の S3 バケットに保存する必要があります。カスタムコンポーネントに docker-compose up コマンドを使用する例については、「Amazon ECR または Docker Hub のパブリックイメージから Docker コンテナを実行する」を参照してください。
- AWS IoT Greengrass をネットワークプロキシの背後で実行する場合、Docker デーモンがプロキシサーバーを使用するように設定してください。
- Docker イメージが Amazon ECR または Docker Hub に保存されている場合、Docker コンポーネントマネージャーコンポーネントを Docker コンテナコンポーネントの従属関係として含めます。コンポーネントをデプロイする前に、コアデバイスの Docker デーモンを起動する必要があります。

  また、イメージ URI をコンポーネントアーティファクトとして含めます。イメージ URI は、次の例で示すように、形式 docker:*registry*/*image*[*:tag|@digest*] である必要があります。

  - プライベート Amazon ECR イメージ: docker:*account-id*.dkr.ecr.*region*.amazonaws.com/*repository*/*image*[*:tag|@digest*]
  - パブリック Amazon ECR イメージ: docker:public.ecr.aws/*repository*/*image*[*:tag|@digest*]

- パブリック Docker Hub イメージ: `docker:name[:tag|@digest]`

パブリックリポジトリに格納されているイメージから Docker コンテナを実行する方法の詳細については、「Amazon ECR または Docker Hub のパブリックイメージから Docker コンテナを実行する」を参照してください。

- Docker イメージが Amazon ECR プライベートリポジトリに格納されている場合、トークン交換のサービスコンポーネントを従属関係として Docker コンテナコンポーネントに含める必要があります。また、Greengrass デバイスのロールは、以下の IAM ポリシー例で示されているように、`ecr:GetAuthorizationToken`、`ecr:BatchGetImage`、`ecr:GetDownloadUrlForLayer` アクションを許可する必要があります。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "ecr:GetAuthorizationToken",
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

Amazon ECR プライベートリポジトリに格納されているイメージから Docker コンテナを実行する方法の情報については、「Amazon ECR のプライベートイメージから Docker コンテナを実行」を参照してください。

- Amazon ECR プライベートリポジトリに格納されている Docker イメージを使用するには、プライベートリポジトリがコアデバイスと同じ AWS リージョン にある必要があります。
- Docker イメージまたは Compose ファイルが S3 バケットに保存されている場合、Greengrass デバイスのロールは `s3:GetObject` 許可を付与して、次の IAM ポリシーの例で示すように、コアデバイスがコンポーネントアーティファクトとしてイメージをダウンロードできるようにする必要があります。

```
{
```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

Amazon S3 に格納されているイメージから Docker コンテナを実行する方法の情報については、「Amazon S3 のイメージから Docker コンテナの実行」を参照してください。

- Docker コンテナコンポーネントのプロセス間通信 (IPC)、AWS 認証情報、ストリームマネージャーを使用するには、Docker コンテナの実行時に追加のオプションを指定する必要があります。詳細については、次を参照してください。
  - Docker コンテナコンポーネントでプロセス間通信の使用
  - Docker コンテナコンポーネント (Linux) でAWS 認証情報の使用
  - Docker コンテナコンポーネント (Linux) でストリームマネージャーの使用

Amazon ECR または Docker Hub のパブリックイメージから Docker コンテナを実行する

このセクションでは、Docker Compose を使用して Amazon ECR と Docker Hub に格納されている Docker イメージから Docker コンテナを実行するカスタムコンポーネントを作成する方法について説明します。

Docker Compose を使用して Docker コンテナを実行するには

1. Docker Compose ファイルを作成して Amazon S3 バケットにアップロードします。Greengrass デバイスのロール がデバイスが Compose ファイルにアクセスできるようにする `s3:GetObject` 許可を付与することを確認します。次の例に示す Compose ファイルの例には、Amazon ECR の Amazon CloudWatch エージェントイメージと Docker Hub の MySQL イメージが含まれています。

```
version: "3"
services:
  cloudwatchagent:
    image: "public.ecr.aws/cloudwatch-agent/cloudwatch-agent:latest"
  mysql:
    image: "mysql:8.0"
```

2. AWS IoT Greengrass コアデバイスに カスタムコンポーネントを作成します。次の例に示すレシピの例には、次のプロパティがあります:

   - 従属関係としての Docker アプリケーション マネージャー コンポーネント。このコンポーネントは、AWS IoT Greengrass がパブリック Amazon ECR と Docker Hub リポジトリからイメージをダウンロードできるようにします。
   - パブリック Amazon ECR リポジトリの Docker イメージを指定するコンポーネントアーティファクト。
   - パブリック Docker Hub リポジトリの Docker イメージを指定するコンポーネントアーティファクト。
   - 実行する Docker イメージのコンテナを含む Docker Compose ファイルを指定するコンポーネントアーティファクト。
   - 指定したイメージからコンテナを作成して起動するため、docker-compose up を使用するライフサイクル実行スクリプト。

   JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.MyDockerComposeComponent",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that uses Docker Compose to run images
 from public Amazon ECR and Docker Hub.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.DockerApplicationManager": {
      "VersionRequirement": "~2.0.0"
    }
  },
  "Manifests": [
    {
```

```
      "Platform": {
        "os": "all"
      },
      "Lifecycle": {
        "run": "docker-compose -f {artifacts:path}/docker-compose.yaml up"
      },
      "Artifacts": [
        {
          "URI": "docker:public.ecr.aws/cloudwatch-agent/cloudwatch-
agent:latest"
        },
        {
          "URI": "docker:mysql:8.0"
        },
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/folder/docker-compose.yaml"
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.MyDockerComposeComponent
ComponentVersion: '1.0.0'
ComponentDescription: 'A component that uses Docker Compose to run images from
 public Amazon ECR and Docker Hub.'
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.DockerApplicationManager:
    VersionRequirement: ~2.0.0
Manifests:
  - Platform:
      os: all
    Lifecycle:
        run: docker-compose -f {artifacts:path}/docker-compose.yaml up
    Artifacts:
      - URI: "docker:public.ecr.aws/cloudwatch-agent/cloudwatch-agent:latest"
      - URI: "docker:mysql:8.0"
      - URI: "s3://DOC-EXAMPLE-BUCKET/folder/docker-compose.yaml"
```

> **Note**
>
> Docker コンテナコンポーネントのプロセス間通信 (IPC)、AWS 認証情報、ストリームマネージャーを使用するには、Docker コンテナの実行時に追加のオプションを指定する必要があります。詳細については、次を参照してください。
>
> - Docker コンテナコンポーネントでプロセス間通信の使用
> - Docker コンテナコンポーネント (Linux) でAWS 認証情報の使用
> - Docker コンテナコンポーネント (Linux) でストリームマネージャーの使用

3. コンポーネントをテストしてが正常に作動することを確認します。

   > **Important**
   >
   > コンポーネントをデプロイする前に、Docker デーモンをインストールして起動する必要があります。

   コンポーネントをローカルにデプロイした後、Docker コンテナ ls コマンドを実行してコンテナが実行されていることを確認できます。

   ```
   docker container ls
   ```

4. コンポーネントの準備ができたら、コンポーネントを AWS IoT Greengrass にアップロードして他のコアデバイスにデプロイします。詳細については、「コアデバイスにデプロイするコンポーネントをパブリッシュ」を参照してください。

Amazon ECR のプライベートイメージから Docker コンテナを実行

このセクションでは、Amazon ECR のプライベートリポジトリに格納されている Docker イメージから、Docker コンテナを実行するカスタムコンポーネントを作成する方法について説明します。

Docker コンテナを実行するには

1. AWS IoT Greengrass コアデバイスに カスタムコンポーネントを作成します。次のプロパティが含まれる次のレシピの例を使用します:

- 従属関係としての Docker アプリケーション マネージャー コンポーネント。このコンポーネントは、AWS IoT Greengrass がプライベートリポジトリからイメージをダウンロードするための認証情報を管理できるようにします。
- 従属関係としてのトークン交換のサービスコンポーネント。このコンポーネントは、AWS IoT Greengrass が Amazon ECR とやり取りするための AWS 認証情報を取得できるようにします。
- プライベート Amazon ECR リポジトリの Docker イメージを指定するコンポーネントアーティファクト。
- イメージからコンテナを作成して起動するため、docker 実行を使用するライフサイクル実行スクリプト。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.MyPrivateDockerComponent",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that runs a Docker container from a
 private Amazon ECR image.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.DockerApplicationManager": {
      "VersionRequirement": "~2.0.0"
    },
    "aws.greengrass.TokenExchangeService": {
      "VersionRequirement": "~2.0.0"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "all"
      },
      "Lifecycle": {
        "run": "docker run account-
id.dkr.ecr.region.amazonaws.com/repository[:tag|@digest]"
      },
      "Artifacts": [
        {
```

```
          "URI": "docker:account-
id.dkr.ecr.region.amazonaws.com/repository[:tag|@digest]"
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.MyPrivateDockerComponent
ComponentVersion: '1.0.0'
ComponentDescription: 'A component that runs a Docker container from a private
 Amazon ECR image.'
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.DockerApplicationManager:
    VersionRequirement: ~2.0.0
  aws.greengrass.TokenExchangeService:
    VersionRequirement: ~2.0.0
Manifests:
  - Platform:
      os: all
    Lifecycle:
        run: docker run account-id.dkr.ecr.region.amazonaws.com/repository[:tag|
@digest]
    Artifacts:
      - URI: "docker:account-id.dkr.ecr.region.amazonaws.com/repository[:tag|
@digest]"
```

Note

Docker コンテナコンポーネントのプロセス間通信 (IPC)、AWS 認証情報、ストリームマネージャーを使用するには、Docker コンテナの実行時に追加のオプションを指定する必要があります。詳細については、次を参照してください。

- Docker コンテナコンポーネントでプロセス間通信の使用
- Docker コンテナコンポーネント (Linux) でAWS 認証情報の使用

- Docker コンテナコンポーネント (Linux) でストリームマネージャーの使用

2. コンポーネントをテストしてが正常に作動することを確認します。

> **Important**
> コンポーネントをデプロイする前に、Docker デーモンをインストールして起動する必要があります。

コンポーネントをローカルにデプロイした後、Docker コンテナ ls コマンドを実行してコンテナが実行されていることを確認できます。

```
docker container ls
```

3. コンポーネントを AWS IoT Greengrass にアップロードして、他のコアデバイスにデプロイします。詳細については、「コアデバイスにデプロイするコンポーネントをパブリッシュ」を参照してください。

Amazon S3 のイメージから Docker コンテナの実行

このセクションでは、Amazon S3 に格納されている Docker イメージのコンポーネントに Docker コンテナを実行する方法について説明します。

Amazon S3 のイメージのコンポーネントに Docker コンテナを実行するには

1. docker 保存コマンドを実行して、Docker コンテナのバックアップを作成します。このバックアップは、AWS IoT Greengrass でコンテナを実行するコンポーネントアーティファクトとして提供します。*hello-world*をイメージの名前に置き換えて、*hello-world.tar*を作成するアーカイブファイルの名前に置き換えます。

```
docker save hello-world > artifacts/com.example.MyDockerComponent/1.0.0/hello-world.tar
```

2. AWS IoT Greengrass コアデバイスに カスタムコンポーネントを作成します。次のプロパティが含まれる次のレシピの例を使用します:

   - アーカイブから Docker イメージをロードするため、docker ロードを使用するライフサイクルインストール スクリプト。

- イメージからコンテナを作成して起動するため、docker 実行を使用するライフサイクル実行スクリプト。--rm オプションは、コンテナの終了時にコンテナのクリーンアップが実行します。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.MyS3DockerComponent",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that runs a Docker container from an
 image in an S3 bucket.",
  "ComponentPublisher": "Amazon",
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": {
          "Script": "docker load -i {artifacts:path}/hello-world.tar"
        },
        "run": {
          "Script": "docker run --rm hello-world"
        }
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.MyS3DockerComponent
ComponentVersion: '1.0.0'
ComponentDescription: 'A component that runs a Docker container from an image in
 an S3 bucket.'
ComponentPublisher: Amazon
Manifests:
  - Platform:
```

```
      os: linux
    Lifecycle:
      install:
        Script: docker load -i {artifacts:path}/hello-world.tar
      run:
        Script: docker run --rm hello-world
```

> **Note**
>
> Docker コンテナコンポーネントのプロセス間通信 (IPC)、AWS 認証情報、ストリームマネージャーを使用するには、Docker コンテナの実行時に追加のオプションを指定する必要があります。詳細については、次を参照してください。
>
> - Docker コンテナコンポーネントでプロセス間通信の使用
> - Docker コンテナコンポーネント (Linux) でAWS 認証情報の使用
> - Docker コンテナコンポーネント (Linux) でストリームマネージャーの使用

3. コンポーネントをテストしてが正常に作動することを確認します。

   コンポーネントをローカルにデプロイした後、Docker コンテナ ls コマンドを実行してコンテナが実行されていることを確認できます。

   ```
   docker container ls
   ```

4. コンポーネントの準備ができたら、S3 バケットに Docker イメージアーカイブをアップロードし、コンポーネントレシピに URI を追加します。次に、コンポーネントを AWS IoT Greengrass にアップロードして他のコアデバイスに展開できます。詳細については、「コアデバイスにデプロイするコンポーネントをパブリッシュ」を参照してください。

   完了したら、コンポーネントレシピは次の例のようになります。

   JSON

   ```
   {
     "RecipeFormatVersion": "2020-01-25",
     "ComponentName": "com.example.MyS3DockerComponent",
     "ComponentVersion": "1.0.0",
     "ComponentDescription": "A component that runs a Docker container from an
    image in an S3 bucket.",
   ```

```
  "ComponentPublisher": "Amazon",
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": {
          "Script": "docker load -i {artifacts:path}/hello-world.tar"
        },
        "run": {
          "Script": "docker run --rm hello-world"
        }
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.MyDockerComponent/1.0.0/hello-world.tar"
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.MyS3DockerComponent
ComponentVersion: '1.0.0'
ComponentDescription: 'A component that runs a Docker container from an image in
 an S3 bucket.'
ComponentPublisher: Amazon
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install:
        Script: docker load -i {artifacts:path}/hello-world.tar
      run:
        Script: docker run --rm hello-world
    Artifacts:
```

```
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.MyDockerComponent/1.0.0/hello-world.tar
```

Docker コンテナコンポーネントでプロセス間通信の使用

AWS IoT Device SDK で Greengrass プロセス間通信 (IPC) ライブラリを使用して Greengrass nucleus、他の Greengrass コンポーネント、AWS IoT Core と通信できます。詳細については、「AWS IoT Device SDK を使用して Greengrass nucleus、その他のコンポーネント、および AWS IoT Core と通信する」を参照してください。

Docker コンテナコンポーネントで IPC を使用するには、次のパラメータで Docker コンテナを実行する必要があります。

- IPC ソケットをコンテナに取り付けます。Greengrass nucleus は、`AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT` 環境変数で IPC ソケットファイルパスを提供します。
- `SVCUID` と `AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT` の環境変数を Greengrass nucleus がコンポーネントに提供する値に設定します。コンポーネントは、これらの環境変数を使用して Greengrass nucleus への接続を認証します。

Example レシピの例: AWS IoT Core (Python) に MQTT メッセージをパブリッシュ

次のレシピは、MQTT メッセージを AWS IoT Core にパブリッシュする Docker コンテナコンポーネントの例を定義します。このレシピには以下のプロパティがあります。

- `accessControl` コンポーネントに対して、すべてのトピックについて MQTT メッセージを AWS IoT Core にパブリッシュすることを許可する承認ポリシー。詳細については、「コンポーネントに IPC オペレーションの実行を許可する」と「AWS IoT Core MQTT IPC 認証」を参照してください。
- Amazon S3 の Docker イメージを TAR アーカイブとして指定するコンポーネントアーティファクト。
- TAR アーカイブから Docker イメージをロードするライフサイクル インストール スクリプト。
- イメージから Docker コンテナを実行するライフサイクル実行スクリプト。Docker 実行コマンドは次の引数がありります。
  - `-v` 引数は Greengrass IPC ソケットをコンテナにマウントします。
  - 最初の 2 つの `-e` 引数は、Docker コンテナに必要な環境変数を設定します。

- 追加の `-e` 引数は、この例で使用される環境変数を設定します。
- `--rm` 引数は、終了時にコンテナのクリーンアップを実行します。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.python.docker.PublishToIoTCore",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Uses interprocess communication to publish an MQTT
 message to IoT Core.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "topic": "test/topic/java",
      "message": "Hello, World!",
      "qos": "1",
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.python.docker.PublishToIoTCore:pubsub:1": {
            "policyDescription": "Allows access to publish to IoT Core on all
 topics.",
            "operations": [
              "aws.greengrass#PublishToIoTCore"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "all"
      },
      "Lifecycle": {
        "install": "docker load -i {artifacts:path}/publish-to-iot-core.tar",
        "run": "docker run -v $AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT:
$AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT -e SVCUID -e
```

```
 AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT -e MQTT_TOPIC=
\"{configuration:/topic}\" -e MQTT_MESSAGE=\"{configuration:/message}\" -e MQTT_QOS=
\"{configuration:/qos}\" --rm publish-to-iot-core"
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.python.docker.PublishToIoTCore/1.0.0/publish-to-iot-core.tar"
        }
      ]
    }
  ]
}
```

YAML

```
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.python.docker.PublishToIoTCore
ComponentVersion: 1.0.0
ComponentDescription: Uses interprocess communication to publish an MQTT message to
 IoT Core.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    topic: 'test/topic/java'
    message: 'Hello, World!'
    qos: '1'
    accessControl:
      aws.greengrass.ipc.mqttproxy:
        'com.example.python.docker.PublishToIoTCore:pubsub:1':
          policyDescription: Allows access to publish to IoT Core on all topics.
          operations:
            - 'aws.greengrass#PublishToIoTCore'
          resources:
            - '*'
Manifests:
  - Platform:
      os: all
    Lifecycle:
      install: 'docker load -i {artifacts:path}/publish-to-iot-core.tar'
      run: |
        docker run \
```

```
          -v $AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT:
$AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT \
          -e SVCUID \
          -e AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT \
          -e MQTT_TOPIC="{configuration:/topic}" \
          -e MQTT_MESSAGE="{configuration:/message}" \
          -e MQTT_QOS="{configuration:/qos}" \
          --rm publish-to-iot-core
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.python.docker.PublishToIoTCore/1.0.0/publish-to-iot-core.tar
```

Docker コンテナコンポーネント (Linux) でAWS 認証情報の使用

トークン交換のサービスコンポーネントを使用して Greengrass コンポーネントの AWS サービスとやり取りできます。このコンポーネントは、ローカル コンテナ サーバーを使用してコアデバイスのトークン交換ロールの AWS 認証情報を提供します。詳細については、「AWS サービスとやり取り」を参照してください。

> **Note**
>
> このセクションの例は Linux コアデバイスにのみ使えます。

Docker コンテナコンポーネントのトークン交換サービスの AWS 認証情報を使用するには、次のパラメータで Docker コンテナを実行する必要があります。

- `--network=host` 引数を使用して、ホストネットワークへのアクセスを提供します。このオプションは、Docker コンテナがローカルトークン交換サービスに接続して AWS 認証情報を取得できるようにします。この引数は、Linux 用 Docker にのみ機能します。

> **Warning**
>
> このオプションは、コンテナがホストのすべてのローカル ネットワーク インターフェイスにアクセスできるようにするため、このオプションは、ホストネットワークにこのアクセスなしで Docker コンテナを実行した場合よりも安全性が低くなります。このオプションを使用する Docker コンテナコンポーネントを開発して実行するときに、この点に注意

してください。詳細については、「Docker マニュアル」の「ネットワーク: ホスト」を参照してください。

- `AWS_CONTAINER_CREDENTIALS_FULL_URI` と `AWS_CONTAINER_AUTHORIZATION_TOKEN` の環境変数を Greengrass nucleus がコンポーネントに提供する値に設定します。AWSSDK はこれらの環境変数を使用して AWS 認証情報を取得します。

Example レシピの例: Docker コンテナコンポーネント (Python) で S3 バケットを一覧表示

次のレシピは、AWS アカウント の S3 バケットを一覧表示する Docker コンテナコンポーネントの例を定義します。このレシピには以下のプロパティがあります。

- 従属関係としてのトークン交換のサービスコンポーネント。この依存関係は、コンポーネントが AWS 認証情報を取得して他の AWS サービスとやり取りできるようにします。
- Amazon S3 の Docker イメージを TAR アーカイブとして指定するコンポーネントアーティファクト。
- TAR アーカイブから Docker イメージをロードするライフサイクル インストール スクリプト。
- イメージから Docker コンテナを実行するライフサイクル実行スクリプト。Docker 実行コマンドは次の引数がありります。
  - `--network=host` 引数は、コンテナがホストネットワークにアクセスを提供するため、コンテナがトークン交換サービスに接続できます。
  - `-e` 引数は、Docker コンテナの必要な環境変数を設定します。
  - `--rm` 引数は、終了時にコンテナのクリーンアップを実行します。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.python.docker.ListS3Buckets",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Uses the token exchange service to lists your S3
 buckets.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.TokenExchangeService": {
      "VersionRequirement": "^2.0.0",
      "DependencyType": "HARD"
```

```
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "docker load -i {artifacts:path}/list-s3-buckets.tar",
        "run": "docker run --network=host -e AWS_CONTAINER_AUTHORIZATION_TOKEN -e
 AWS_CONTAINER_CREDENTIALS_FULL_URI --rm list-s3-buckets"
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.python.docker.ListS3Buckets/1.0.0/list-s3-buckets.tar"
        }
      ]
    }
  ]
}
```

YAML

```
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.python.docker.ListS3Buckets
ComponentVersion: 1.0.0
ComponentDescription: Uses the token exchange service to lists your S3 buckets.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.TokenExchangeService:
    VersionRequirement: ^2.0.0
    DependencyType: HARD
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: 'docker load -i {artifacts:path}/list-s3-buckets.tar'
      run: |
        docker run \
          --network=host \
          -e AWS_CONTAINER_AUTHORIZATION_TOKEN \
          -e AWS_CONTAINER_CREDENTIALS_FULL_URI \
```

```
            --rm list-s3-buckets
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.python.docker.ListS3Buckets/1.0.0/list-s3-buckets.tar
```

Docker コンテナコンポーネント (Linux) でストリームマネージャーの使用

ストリーム マネージャー コンポーネント を使用して Greengrass コンポーネントのデータストリームを管理します。このコンポーネントは、データストリーム処理と大量の IoT データを AWS クラウド に転送できるようにします。AWS IoT Greengrass は、ストリームマネージャーコンポーネントとやり取りするために使用するストリームマネージャー SDK を提供します。詳細については、「Greengrass コアデバイスでのデータストリームの管理」を参照してください。

> **Note**
>
> このセクションの例は Linux コアデバイスにのみ使えます。

Docker コンテナコンポーネントでストリームマネージャー SDK を使用するには、次のパラメータで Docker コンテナを実行する必要があります。

- `--network=host` 引数を使用して、ホストネットワークへのアクセスを提供します。このオプションは、Docker コンテナがローカル TLS 接続を介してストリームマネージャーコンポーネントとやり取りできるようにします。この引数は Linux 用 Docker にのみ使えます

  > **Warning**
  >
  > このオプションは、コンテナがホストのすべてのローカル ネットワーク インターフェイスにアクセスできるようにするため、このオプションは、ホストネットワークにこのアクセスなしで Docker コンテナを実行した場合よりも安全性が低くなります。このオプションを使用する Docker コンテナコンポーネントを開発して実行するときに、この点に注意してください。詳細については、「Docker マニュアル」の「ネットワーク: ホスト」を参照してください。

- 認証を要求する (デフォルト動作) ようにストリーム マネージャー コンポーネントを設定する場合、`AWS_CONTAINER_CREDENTIALS_FULL_URI` 環境変数を Greengrass nucleus がコンポーネントに提供する値に設定します。詳細については、「ストリームマネージャーの設定」を参照してください。

- ストリーム マネージャー コンポーネントがデフォルト以外のポートを使用するように設定する場合は、プロセス間通信 (IPC) を使用してストリーム マネージャー コンポーネント設定からポートを取得します。IPC を使用するには、追加オプションを指定して Docker コンテナを実行する必要があります。詳細については、次を参照してください。
  - アプリケーションコードでストリームマネージャーに接続
  - Docker コンテナコンポーネントでプロセス間通信の使用

Example レシピの例: Docker コンテナコンポーネント (Python) で S3 バケットにファイルのストリーミング

次のレシピは、ファイルを作成して S3 バケットにストリーミングする Docker コンテナコンポーネントの例を定義します。このレシピには以下のプロパティがあります。

- 従属関係としてのストリーム マネージャー コンポーネント。この依存関係により、コンポーネントはストリームマネージャー SDK を使用してストリームマネージャーコンポーネントとやり取りできます。
- Amazon S3 の Docker イメージを TAR アーカイブとして指定するコンポーネントアーティファクト。
- TAR アーカイブから Docker イメージをロードするライフサイクル インストール スクリプト。
- イメージから Docker コンテナを実行するライフサイクル実行スクリプト。Docker 実行コマンドは次の引数がありります。
  - `--network=host` 引数は、コンテナにホストネットワークへのアクセスを提供するため、コンテナがストリームマネージャーコンポーネントに接続できます。
  - 最初の `-e` 引数は、Docker コンテナの必要な `AWS_CONTAINER_AUTHORIZATION_TOKEN` 環境変数を設定します。
  - 追加の `-e` 引数は、この例で使用される環境変数を設定します。
  - `-v` 引数は、コンポーネントの作業フォルダをコンテナにマウントします。この例では、ストリームマネージャーを使用して、Amazon S3 にアップロードするファイルを作業フォルダに作成します。
  - `--rm` 引数は、終了時にコンテナのクリーンアップを実行します。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
```

```
  "ComponentName": "com.example.python.docker.StreamFileToS3",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Creates a text file and uses stream manager to stream the
 file to S3.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.StreamManager": {
      "VersionRequirement": "^2.0.0",
      "DependencyType": "HARD"
    }
  },
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "bucketName": ""
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "docker load -i {artifacts:path}/stream-file-to-s3.tar",
        "run": "docker run --network=host -e AWS_CONTAINER_AUTHORIZATION_TOKEN
 -e BUCKET_NAME=\"{configuration:/bucketName}\" -e WORK_PATH=\"{work:path}\" -v
 {work:path}:{work:path} --rm stream-file-to-s3"
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.python.docker.StreamFileToS3/1.0.0/stream-file-to-s3.tar"
        }
      ]
    }
  ]
}
```

YAML

```
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.python.docker.StreamFileToS3
ComponentVersion: 1.0.0
```

```
ComponentDescription: Creates a text file and uses stream manager to stream the file
 to S3.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.StreamManager:
    VersionRequirement: ^2.0.0
    DependencyType: HARD
ComponentConfiguration:
  DefaultConfiguration:
    bucketName: ''
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: 'docker load -i {artifacts:path}/stream-file-to-s3.tar'
      run: |
        docker run \
          --network=host \
          -e AWS_CONTAINER_AUTHORIZATION_TOKEN \
          -e BUCKET_NAME="{configuration:/bucketName}" \
          -e WORK_PATH="{work:path}" \
          -v {work:path}:{work:path} \
          --rm stream-file-to-s3
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.python.docker.StreamFileToS3/1.0.0/stream-file-to-s3.tar
```

AWS IoT Greengrass コンポーネントレシピのリファレンス

コンポーネントレシピは、コンポーネントの詳細、依存関係、アーティファクト、およびライフサイクルを定義するファイルです。コンポーネントのライフサイクルでは、コンポーネントのインストール、実行、シャットダウンなどのために実行するコマンドを指定します。AWS IoT Greengrass コアは、レシピで定義したライフサイクルを使用して、コンポーネントをインストールして実行します。AWS IoT Greengrass サービスは、コンポーネントをデプロイするときに、コアデバイスにデプロイする依存関係とアーティファクトを特定するためにレシピを使用します。

レシピでは、コンポーネントがサポートしているプラットフォームごとに、固有の依存関係とライフサイクルを定義できます。この機能を使用すれば、要件が異なる複数のプラットフォームを持つデバイスにコンポーネントをデプロイすることができます。また、この機能を使用することで、AWS IoT Greengrass がサポートしていないデバイスにコンポーネントをインストールしないようにすることもできます。

各レシピには、マニフェストのリストが含まれます。各マニフェストは、プラットフォームがそれらの要件を満たすコアデバイスに使用する一連のプラットフォーム要件、ライフサイクル、そしてアーティファクトを指定します。コアデバイスは、デバイスが満たすプラットフォーム要件が含まれる最初のマニフェストを使用します。プラットフォーム要件のないマニフェストを指定すれば、すべてのコアデバイスと一致します。

マニフェストにないグローバルライフサイクルを指定することもできます。グローバルライフサイクルでは、ライフサイクルのサブセクションを識別する選択キーを使用することができます。その後、マニフェスト内でこれらの選択キーを指定することで、マニフェストのライフサイクルに加えて、グローバルライフサイクルのこれらのセクションも使用することができます。コアデバイスは、マニフェストがライフサイクルを定義していない場合にのみ、マニフェストの選択キーを使用します。マニフェストの `all` 選択を使用すると、選択キーのないグローバルライフサイクルのセクションに一致させることができます。

AWS IoT Greengrass Core ソフトウェアは、コアデバイスと一致するマニフェストを選択した後、使用するライフサイクルステップを識別するために以下を実行します。

- 選択したマニフェストがライフサイクルを定義している場合、コアデバイスはそのライフサイクルを使用します。
- 選択したマニフェストがライフサイクルを定義していない場合、コアデバイスはグローバルサイクルを使用します。コアデバイスは、グローバルライフサイクルのどのセクションを使用するかを特定するために以下を実行します。
  - マニフェストが選択キーを定義している場合、コアデバイスはマニフェストの選択キーが含まれるグローバルライフサイクルのセクションを使用します。
  - マニフェストが選択キーを定義していない場合、コアデバイスはマニフェストの選択キーが含まれないグローバルライフサイクルのセクションを使用します。この動作は、`all` 選択を定義するマニフェストと同じです。

> **Important**
>
> コアデバイスは、コンポーネントをインストールするにあたり、マニフェストのプラットフォーム要件を少なくとも 1 つ満たす必要があります。コアデバイスに一致するマニフェストがない場合、AWS IoT Greengrass Core ソフトウェアはコンポーネントをインストールせず、デプロイは失敗します。

レシピは JSON または YAML 形式で定義できます。レシピの例セクションには、各形式のレシピが含まれています。

トピック

レシピの検証

Greengrass は、コンポーネントバージョンの作成時に JSON または YAML コンポーネントレシピを検証します。この recipe 検証では、JSON または YAML コンポーネント recipe に一般的なエラーがないかをチェックして、潜在的なデプロイの問題を防ぎます。検証では、レシピに一般的なエラー (カンマ、中括弧、フィールドの欠落など) がないかチェックし、レシピの形式が適切であることを確認します。

recipe 検証エラーメッセージが表示された場合は、欠落しているカンマ、中括弧、またはフィールドがないか recipe を確認してください。recipe 形式 を確認して、フィールドが欠落していないことを確認します。

レシピの形式

コンポーネントのレシピを定義するときには、レシピドキュメントで次の情報を指定します。YAML と JSON 形式のレシピにも同じ構造が適用されます。

`RecipeFormatVersion`

レシピのテンプレートバージョン。次のオプションを選択します。

- `2020-01-25`

`ComponentName`

このレシピが定義するコンポーネントの名前。コンポーネントの名前は、各リージョンの AWS アカウント で一意である必要があります。

> **ヒント**
>
> - 逆ドメイン名の形式にすれば、社内での名前のコリジョンを回避することができます。例えば、会社が `example.com` を所有しており、太陽エネルギー

プロジェクトに取り組んでいる場合は、Hello World コンポーネントの名前を `com.example.solar.HelloWorld` にすることができます。こうすることで、会社内のコンポーネント名のコリジョンを回避できます。

- コンポーネント名に `aws.greengrass` プレフィックスを使用することは避けてください。AWS IoT Greengrass は提供するパブリックコンポーネントでこのプレフィックスを使用します。パブリックコンポーネントと同じ名前を選択すると、パブリックコンポーネントがあなたのコンポーネントに置き換えられます。そうすると、AWS IoT Greengrass がそのパブリックコンポーネントに依存するコンポーネントをデプロイするときに、パブリックコンポーネントの代わりにあなたのコンポーネントを提供してしまいます。この機能を使用すると、パブリックコンポーネントの動作を上書きできますが、パブリックコンポーネントの上書きを意図していない場合には、他のコンポーネントも破損させてしまう可能性があります。

`ComponentVersion`

コンポーネントのバージョン。メジャー、マイナー、パッチ値の最大値は 99999 です。

> **Note**
>
> AWS IoT Greengrass はコンポーネントのセマンティックバージョンを使用します。セマンティックバージョンは、major.minor.patch といった番号システムに準拠します。例えば、バージョン `1.0.0` は、コンポーネントの最初のメジャーリリースを表しています。詳細については、「セマンティックバージョンの仕様」を参照してください。

`ComponentDescription`

(オプション) コンポーネントの説明。

`ComponentPublisher`

コンポーネントのパブリッシャーまたは作成者。

`ComponentConfiguration`

(オプション) コンポーネントの設定またはパラメータを定義するオブジェクト。デフォルト設定を定義しておけば、コンポーネントをデプロイするときに、コンポーネントに提供する設定オブジェクトを指定することができます。コンポーネント設定は、ネストされたパラメータと構造をサポートします。このオブジェクトには、次の情報が含まれます。

`DefaultConfiguration`

コンポーネントのデフォルト設定を定義するオブジェクト。このオブジェクトの構造を定義します。

> **Note**
>
> AWS IoT Greengrass は設定値に JSON を使用します。JSON は数値タイプを指定しますが、整数と浮動小数点数を区別しません。その結果、AWS IoT Greengrass で設定値が浮動小数点数に変換されることがあります。コンポーネントが正しいデータタイプを使用することを確認するには、数値の設定値を文字列として定義することをお勧めします。次に、整数または浮動小数点としてコンポーネントでパースします。これにより、設定値が設定とコアデバイスに対して同じタイプであることを保証します。

`ComponentDependencies`

(オプション) コンポーネントのコンポーネント依存関係を定義するオブジェクトのディクショナリ。各オブジェクトのキーは、コンポーネントの依存関係の名前を識別します。AWS IoT Greengrass はコンポーネントをインストールするときに、コンポーネントの依存関係をインストールします。AWS IoT Greengrass はコンポーネントを起動する前に、依存関係が開始されるまで待機します。各オブジェクトには、次の情報が含まれます:

`VersionRequirement`

この依存関係との互換性のあるコンポーネントバージョンを定義する npm スタイルのセマンティックバージョン制約。1 つのバージョンまたはバージョンの範囲を指定できます。詳細については、「npm セマンティックバージョン計算ツール」を参照してください。

`DependencyType`

(オプション) 依存関係のタイプ。次のオプションから選択します。

- `SOFT` - 依存関係が状態を変化させても、コンポーネントは再起動しません。
- `HARD` - 依存関係が状態を変化させると、コンポーネントは再起動します。

デフォルトは `HARD` です。

`ComponentType`

(オプション) コンポーネントのタイプ。

> **Note**
>
> レシピでコンポーネントタイプを指定することは推奨しません。AWS IoT Greengrass がコンポーネントを作成するときにタイプを設定します。

タイプは、次のいずれかになります。

- `aws.greengrass.generic` - コンポーネントはコマンドを実行するか、アーティファクトを提供します。
- `aws.greengrass.lambda` – コンポーネントは Lambda ランチャーコンポーネントを使用して、 Lambda 関数を実行します。`ComponentSource` パラメータは、このコンポーネントが実行する Lambda 関数の ARN を指定します。

  このオプションは推奨しません。Lambda 関数からコンポーネントを作成するときに、AWS IoT Greengrass によって設定されるからです。詳細については、「AWS Lambda 関数を実行する」を参照してください。
- `aws.greengrass.plugin` – コンポーネントは Greengrass nucleus と同じ Java 仮想マシン (JVM) で実行されます。プラグインコンポーネントをデプロイするか再起動すると、Greengrass nucleus が再起動します。

  プラグインコンポーネントは Greengrass nucleus と同じログファイルを使用します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

  コンポーネントレシピでこのオプションを使用することは推奨しません。これは、Greengrass nucleus と直接やり取りする Java で書かれた AWS から提供されたコンポーネントを意図したものだからです。どのパブリックコンポーネントがプラグインなのかの詳細については、「AWS が提供したコンポーネント」を参照してください。
- `aws.greengrass.nucleus` - nucleus コンポーネント。詳細については、「Greengrass nucleus」を参照してください。

  コンポーネントレシピでこのオプションを使用することは推奨しません。これは、Greengrass nucleus コンポーネントを対象としたもので、AWS IoT Greengrass Core ソフトウェアの最低限の機能のみが提供されるからです。

レシピからコンポーネントを作成する場合は `aws.greengrass.generic`、Lambda 関数からコンポーネントを作成する場合は `aws.greengrass.lambda` がデフォルトです。

詳細については、「コンポーネントタイプ」を参照してください。

`ComponentSource`

(オプション) コンポーネントが実行する Lambda 関数の ARN。

レシピでコンポーネントソースを指定することは推奨しません。Lambda 関数からコンポーネントを作成するときに、AWS IoT Greengrass がこのパラメータを設定します。詳細については、「AWS Lambda 関数を実行する」を参照してください。

`Manifests`

オブジェクトのリストで、各オブジェクトがコンポーネントのライフサイクル、パラメータ、およびプラットフォームに対する要件を定義しています。コアデバイスが複数のマニフェストのプラットフォーム要件と一致する場合、AWS IoT Greengrass は、コアデバイスが一致する最初のマニフェストを使用します。コアデバイスが正しいマニフェストを確実に使用するように、まずはより厳格なプラットフォーム要件を持つマニフェストを定義してください。すべてのプラットフォームに適用されるマニフェストは、リストの最後のマニフェストである必要があります。

> **Important**
>
> コアデバイスは、コンポーネントをインストールするにあたり、マニフェストのプラットフォーム要件を少なくとも 1 つ満たす必要があります。コアデバイスに一致するマニフェストがない場合、AWS IoT Greengrass Core ソフトウェアはコンポーネントをインストールせず、デプロイは失敗します。

各オブジェクトには、次の情報が含まれます:

`Name`

(オプション) このマニフェストが定義するプラットフォームのわかりやすい名前。

このパラメータを省略すると、AWS IoT Greengrass はプラットフォームの `os` と `architecture` からわかりやすい名前を作成します。

`Platform`

(オプション) このマニフェストが適用されるプラットフォームを定義するオブジェクト。すべてのプラットフォームに適用されるマニフェストを定義する場合は、このパラメータを省略します。

このオブジェクトは、コアデバイスが動作するプラットフォームに関するキーと値のペアを指定します。このコンポーネントをデプロイすると、AWS IoT Greengrass Core ソフトウェア

は、これらのキーと値のペアをコアデバイス上のプラットフォーム属性と比較します。AWS IoT Greengrass Core ソフトウェアが常に `os` と `architecture` を定義し、追加の属性を定義する場合もあります。Greengrass nucleus コンポーネントをデプロイするときに、コアデバイスのカスタムプラットフォーム属性を指定することができます。詳細については、「Greengrass nucleus コンポーネント」の「プラットフォームの上書きパラメータ」を参照してください。

キーと値のペアごとに、次のいずれかの値を指定できます。

- `linux` または `windows` などの正確な値。正確な値は、数字または文字で始まっている必要があります。
- *: これはすべての値と一致します。値が存在しない場合でも一致します。
- `/windows|linux/` などの Java スタイルの正規表現。正規表現は、スラッシュ文字 (/) で開始し終了する必要があります。たとえば、正規表現 `/.+/` は空白以外のすべての値と一致します。

このオブジェクトには、次の情報が含まれます。

`os`

(オプション) このマニフェストがサポートするプラットフォームのオペレーティングシステムの名前。一般的なプラットフォームには次の値が含まれます。

- `linux`
- `windows`
- `darwin` (macOS)

`architecture`

(オプション) このマニフェストがサポートするプラットフォームのプロセッサアーキテクチャ。一般的なアーキテクチャには次の値が含まれます。

- `amd64`
- `arm`
- `aarch64`
- `x86`

`architecture.detail`

(オプション) このマニフェストがサポートするプラットフォームのプロセッサアーキテクチャの詳細。一般的なアーキテクチャの詳細には次の値が含まれます。

- `arm61`

- arm71
- arm81

key

(オプション) このマニフェストに対して定義するプラットフォーム属性。*key* をプラットフォーム属性の名前に置き換えます。AWS IoT Greengrass Core ソフトウェアは、このプラットフォーム属性を Greengrass nucleus コンポーネント設定で指定したキーと値のペアと照合します。詳細については、「Greengrass nucleus コンポーネント」の「プラットフォームの上書きパラメータ」を参照してください。

> **Tip**
>
> 逆ドメイン名の形式にすれば、社内での名前のコリジョンを回避することができます。例えば、会社が example.com を所有していて、ラジオプロジェクトで作業している場合には、カスタムプラットフォーム属性に com.example.radio.RadioModule という名前を付けることができます。こうすることで、会社内のプラットフォーム属性名のコリジョンを回避できます。

たとえば、com.example.radio.RadioModule というプラットフォーム属性を定義して、コアデバイスで利用可能な無線モジュールに基づいて、異なるマニフェストを指定することができます。各マニフェストには、異なるハードウェア設定に適用する異なるアーティファクトを含めることができるため、コアデバイスにデプロイするソフトウェアセットを最小限に抑えることができます。

Lifecycle

このマニフェストが定義するプラットフォームに、コンポーネントをインストールして実行する方法を定義するオブジェクトまたは文字列。すべてのプラットフォームに適用されるグローバルライフサイクルを定義することもできます。コアデバイスは、使用するマニフェストでライフサイクルが指定されていない場合にのみ、グローバルライフサイクルを使用します。

> **Note**
>
> このライフサイクルはマニフェスト内で定義します。ここで指定するライフサイクルステップは、このマニフェストが定義するプラットフォームにのみ適用されます。すべてのプラットフォームに適用されるグローバルライフサイクルを定義することもできます。

このオブジェクトまたは文字列には、次の情報が含まれます。

`Setenv`

(オプション) すべてのライフサイクルスクリプトに提供する、環境変数のディクショナリ。これらの環境変数は、各ライフサイクルスクリプトの `Setenv` で上書きすることができます。

`install`

(オプション) コンポーネントのインストール時に実行するスクリプトを定義するオブジェクトまたは文字列。AWS IoT Greengrass Core ソフトウェアは、ソフトウェアが起動するたびに、このライフサイクルステップを実行します。

`install` スクリプトが成功コードで終了すると、コンポーネントは `INSTALLED` 状態に入ります。

このオブジェクトまたは文字列には、次の情報が含まれます。

`Script`

実行するスクリプト。

`RequiresPrivilege`

(オプション) ルート権限でスクリプトを実行できます。このオプションを `true` に設定すると、AWS IoT Greengrass Core ソフトウェアは、このコンポーネントを実行するように設定したシステムユーザーとしてではなく、root としてこのライフサイクルスクリプトを実行します。デフォルトは `false` です。

`Skipif`

(オプション) スクリプトを実行するかどうかを決定するためのチェック。実行可能ファイルがパス上にあるかどうか、あるいはファイルが存在するかどうかを確認するように定義できます。出力が True の場合、AWS IoT Greengrass Core ソフトウェアはこのステップをスキップします。次のいずれかの値を選択します。

- `onpath` *`runnable`* - システムパス上に runnable があるかどうかをチェックします。例えば、Python 3 が利用可能な場合には、**`onpath python3`** を使用してこのライフサイクルステップをスキップします。
- `exists` *`file`* - ファイルが存在するかどうかをチェックします。たとえば、`/tmp/my-configuration.db` が存在する場合には、**`exists /tmp/my-configuration.db`** を使用してこのライフサイクルステップをスキップします。

`Timeout`

(オプション) AWS IoT Greengrass Core ソフトウェアがプロセスを終了する前に、スクリプトが実行できる最大時間 (秒単位)。

デフォルト: 120 秒

`Setenv`

(オプション) スクリプトに提供する環境変数のディクショナリ。これらの環境変数は、`Lifecycle.Setenv` で指定した変数を上書きします。

`run`

(オプション) コンポーネントの開始時に実行するスクリプトを定義するオブジェクトまたは文字列。

このライフサイクルステップが実行されると、コンポーネントは `RUNNING` 状態に入ります。`run` スクリプトが成功コードで終了すると、コンポーネントは `STOPPING` 状態に入ります。`shutdown` スクリプトが指定されている場合は、スクリプトが実行され、指定されていない場合、コンポーネントは `FINISHED`状態になります。

このコンポーネントに依存するコンポーネントは、このライフサイクルステップが実行されたときに開始されます。依存コンポーネントが使用するサービスなどのバックグラウンドプロセスを実行するには、代わりに `startup` ライフサイクルステップを使用します。

`run` ライフサイクルによりコンポーネントをデプロイすると、そのライフサイクルスクリプトが実行された直後から、デプロイが完了したコアデバイスを報告できるようになります。その結果、`run` ライフサイクルスクリプトが実行後すぐに失敗した場合でも、そのデプロイを正常に完了することができます。デプロイステータスを、コンポーネントの開始スクリプトの結果に依存させたい場合は、代わりに `startup` ライフサイクルステップを使用します。

> **Note**
>
> 1 つの `startup` または `run` ライフサイクルのみを定義できます。

このオブジェクトまたは文字列には、次の情報が含まれます。

`Script`

実行するスクリプト。

`RequiresPrivilege`

(オプション) ルート権限でスクリプトを実行できます。このオプションを `true` に設定すると、AWS IoT Greengrass Core ソフトウェアは、このコンポーネントを実行するように設定したシステムユーザーとしてではなく、root としてこのライフサイクルスクリプトを実行します。デフォルトは `false` です。

`Skipif`

(オプション) スクリプトを実行するかどうかを決定するためのチェック。実行可能ファイルがパス上にあるかどうか、あるいはファイルが存在するかどうかを確認するように定義できます。出力が True の場合、AWS IoT Greengrass Core ソフトウェアはこのステップをスキップします。次のいずれかの値を選択します。

- `onpath` *`runnable`* - システムパス上に runnable があるかどうかをチェックします。例えば、Python 3 が利用可能な場合には、**`onpath python3`** を使用してこのライフサイクルステップをスキップします。
- `exists` *`file`* - ファイルが存在するかどうかをチェックします。たとえば、`/tmp/my-configuration.db` が存在する場合には、**`exists /tmp/my-configuration.db`** を使用してこのライフサイクルステップをスキップします。

`Timeout`

(オプション) AWS IoT Greengrass Core ソフトウェアがプロセスを終了する前に、スクリプトが実行できる最大時間 (秒単位)。

デフォルトでは、このライフサイクルステップはタイムアウトしません。このタイムアウトを省略すると、`run` スクリプトが終了するまで実行されます。

`Setenv`

(オプション) スクリプトに提供する環境変数のディクショナリ。これらの環境変数は、`Lifecycle.Setenv` で指定した変数を上書きします。

`startup`

(オプション) コンポーネントの開始時に実行するバックグラウンドプロセスを定義するオブジェクトまたは文字列。

依存関係のあるコンポーネントを起動する前に、正常に終了させる、あるいはコンポーネントのステータスを `RUNNING` に更新するコマンドを実行するには、`startup` を使用します。UpdateState IPC オペレーションを使用して、コンポーネントが終了しないスクリプ

トを開始するERROREDときに、コンポーネントのステータスを RUNNINGまたは に設定します。例えば、/etc/init.d/mysqld start で MySQL プロセスを開始する startup ステップを定義することができます。

このライフサイクルステップが実行されると、コンポーネントは STARTING 状態に入ります。startup スクリプトが成功コードで終了すると、コンポーネントは RUNNING 状態に入ります。その後、依存コンポーネントを開始することができます。

startup ライフサイクルによりコンポーネントをデプロイすると、そのライフサイクルスクリプトが終了したか、もしくは状態を報告した後に、コアデバイスがデプロイの完了を報告できるようになります。言い換えると、すべてのコンポーネントの起動スクリプトが終了したか、あるいは状態を報告するまで、デプロイのステータスは IN_PROGRESS のままです

> **Note**
>
> 1 つの startup または run ライフサイクルのみを定義できます。

このオブジェクトまたは文字列には、次の情報が含まれます。

`Script`

実行するスクリプト。

`RequiresPrivilege`

(オプション) ルート権限でスクリプトを実行できます。このオプションを true に設定すると、AWS IoT Greengrass Core ソフトウェアは、このコンポーネントを実行するように設定したシステムユーザーとしてではなく、root としてこのライフサイクルスクリプトを実行します。デフォルトは false です。

`Skipif`

(オプション) スクリプトを実行するかどうかを決定するためのチェック。実行可能ファイルがパス上にあるかどうか、あるいはファイルが存在するかどうかを確認するように定義できます。出力が True の場合、AWS IoT Greengrass Core ソフトウェアはこのステップをスキップします。次のいずれかの値を選択します。

- onpath *runnable* - システムパス上に runnable があるかどうかをチェックします。例えば、Python 3 が利用可能な場合には、**onpath python3** を使用してこのライフサイクルステップをスキップします。

- `exists` *`file`* - ファイルが存在するかどうかをチェックします。たとえば、`/tmp/my-configuration.db` が存在する場合には、**`exists /tmp/my-configuration.db`** を使用してこのライフサイクルステップをスキップします。

`Timeout`

(オプション) AWS IoT Greengrass Core ソフトウェアがプロセスを終了する前に、スクリプトが実行できる最大時間 (秒単位)。

デフォルト: 120 秒

`Setenv`

(オプション) スクリプトに提供する環境変数のディクショナリ。これらの環境変数は、`Lifecycle.Setenv` で指定した変数を上書きします。

`shutdown`

(オプション) コンポーネントのシャットダウン時に実行するスクリプトを定義するオブジェクトまたは文字列。シャットダウンライフサイクルを使用して、コンポーネントがSTOPPING状態にあるときに実行するコードを実行します。シャットダウンライフサイクルは、 `startup`または `run`スクリプトによって開始されたプロセスを停止するために使用できます。

`startup` でバックグラウンドプロセスを開始した場合は、コンポーネントがシャットダウンするときには `shutdown` ステップを使用してそのプロセスを停止します。例えば、`/etc/init.d/mysqld stop` で MySQL プロセスを停止する `shutdown` ステップを定義することができます。

`shutdown` スクリプトは、コンポーネントが STOPPING状態になった後に実行されます。スクリプトが正常に完了すると、コンポーネントは FINISHED状態になります。

このオブジェクトまたは文字列には、次の情報が含まれます。

`Script`

実行するスクリプト。

`RequiresPrivilege`

(オプション) ルート権限でスクリプトを実行できます。このオプションを `true` に設定すると、AWS IoT Greengrass Core ソフトウェアは、このコンポーネントを実行する

ように設定したシステムユーザーとしてではなく、root としてこのライフサイクルスクリプトを実行します。デフォルトは `false` です。

`Skipif`

(オプション) スクリプトを実行するかどうかを決定するためのチェック。実行可能ファイルがパス上にあるかどうか、あるいはファイルが存在するかどうかを確認するように定義できます。出力が True の場合、AWS IoT Greengrass Core ソフトウェアはこのステップをスキップします。次のいずれかの値を選択します。

- `onpath` *`runnable`* - システムパス上に runnable があるかどうかをチェックします。例えば、Python 3 が利用可能な場合には、**`onpath python3`** を使用してこのライフサイクルステップをスキップします。
- `exists` *`file`* - ファイルが存在するかどうかをチェックします。たとえば、`/tmp/my-configuration.db` が存在する場合には、**`exists /tmp/my-configuration.db`** を使用してこのライフサイクルステップをスキップします。

`Timeout`

(オプション) AWS IoT Greengrass Core ソフトウェアがプロセスを終了する前に、スクリプトが実行できる最大時間 (秒単位)。

デフォルト: 15 秒。

`Setenv`

(オプション) スクリプトに提供する環境変数のディクショナリ。これらの環境変数は、`Lifecycle.Setenv` で指定した変数を上書きします。

`recover`

(オプション) コンポーネントにエラーが発生したときに実行するスクリプトを定義するオブジェクトまたは文字列。

この手順は、コンポーネントが ERRORED 状態になったときに実行されます。コンポーネントが正常に回復しないで 3 回 ERRORED になると、コンポーネントは BROKEN 状態に変わります。BROKEN コンポーネントを修復するには、もう一度デプロイする必要があります。

このオブジェクトまたは文字列には、次の情報が含まれます。

`Script`

実行するスクリプト。

`RequiresPrivilege`

(オプション) ルート権限でスクリプトを実行できます。このオプションを `true` に設定すると、AWS IoT Greengrass Core ソフトウェアは、このコンポーネントを実行するように設定したシステムユーザーとしてではなく、root としてこのライフサイクルスクリプトを実行します。デフォルトは `false` です。

`Skipif`

(オプション) スクリプトを実行するかどうかを決定するためのチェック。実行可能ファイルがパス上にあるかどうか、あるいはファイルが存在するかどうかを確認するように定義できます。出力が True の場合、AWS IoT Greengrass Core ソフトウェアはこのステップをスキップします。次のいずれかの値を選択します。

- `onpath` *`runnable`* - システムパス上に runnable があるかどうかをチェックします。例えば、Python 3 が利用可能な場合には、**`onpath python3`** を使用してこのライフサイクルステップをスキップします。
- `exists` *`file`* - ファイルが存在するかどうかをチェックします。たとえば、`/tmp/my-configuration.db` が存在する場合には、**`exists /tmp/my-configuration.db`** を使用してこのライフサイクルステップをスキップします。

`Timeout`

(オプション) AWS IoT Greengrass Core ソフトウェアがプロセスを終了する前に、スクリプトが実行できる最大時間 (秒単位)。

デフォルト: 60 秒

`Setenv`

(オプション) スクリプトに提供する環境変数のディクショナリ。これらの環境変数は、`Lifecycle.Setenv` で指定した変数を上書きします。

`bootstrap`

(オプション) AWS IoT Greengrass Core ソフトウェアまたはコアデバイスの再起動が必要となるスクリプトを定義するオブジェクトまたは文字列。これにより、オペレーティングシステムの更新やランタイムの更新などをインストールした後に再起動するコンポーネントを開発することができます。

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアまたはデバイスを再起動する必要のない更新や依存関係をインストールする場合は、インストールライフサイクルを使用します。

このライフサイクルステップは、AWS IoT Greengrass Core ソフトウェアがコンポーネントをデプロイするときに、以下のケースの場合にインストールライフサイクルステップの前に実行されます。

- コンポーネントがコアデバイスに初めてデプロイされる場合。
- コンポーネントのバージョンが変更される場合。
- コンポーネント設定が更新されたことで、ブートストラップスクリプトが変更される場合。

AWS IoT Greengrass Core ソフトウェアが、デプロイにブートストラップステップが含まれるすべてのコンポーネントのブートストラップステップを完了すると、ソフトウェアは再起動します。

> **Important**
>
> AWS IoT Greengrass Core ソフトウェアまたはコアデバイスを再起動するには、AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定する必要があります。AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定しなかった場合、ソフトウェアは再起動しません。詳細については、「Greengrass nucleus をシステムサービスとして設定する」を参照してください。

このオブジェクトまたは文字列には、次の情報が含まれます。

`BootstrapOnRollback`

> **Note**
>
> この機能を有効にするBootstrapOnRollbackと、 は、失敗したターゲットデプロイの一部としてブートストラップライフサイクルステップを完了ま

たは実行しようとしたコンポーネントに対してのみ実行されます。この機能は、Greengrass nucleus バージョン 2.12.0 以降で使用できます。

(オプション) ロールバックデプロイの一部として、ブートストラップライフサイクルステップを実行できます。このオプションを に設定するとtrue、ロールバックデプロイ内で定義されたブートストラップライフサイクルステップが実行されます。デプロイが失敗すると、ロールバックデプロイ中に以前のバージョンのコンポーネントブートストラップライフサイクルが再度実行されます。

デフォルトは `false` です。

`Script`

実行するスクリプト。このスクリプトの終了コードは、再起動の指示を定義しています。次の終了コードを使用します。

- `0` - AWS IoT Greengrass Core ソフトウェアまたはコアデバイスを再起動します。AWS IoT Greengrass Core ソフトウェアは、すべてのコンポーネントのブートストラップ後に再起動します。
- `100` - AWS IoT Greengrass Core ソフトウェアの再起動をリクエストします。
- `101` - コアデバイスの再起動をリクエストします。

終了コード 100 ~ 199 は特別動作向けに予約済みです。その他の終了コードは、スクリプトエラーを表します。

`RequiresPrivilege`

(オプション) ルート権限でスクリプトを実行できます。このオプションを `true` に設定すると、AWS IoT Greengrass Core ソフトウェアは、このコンポーネントを実行するように設定したシステムユーザーとしてではなく、root としてこのライフサイクルスクリプトを実行します。デフォルトは `false` です。

`Timeout`

(オプション) AWS IoT Greengrass Core ソフトウェアがプロセスを終了する前に、スクリプトが実行できる最大時間 (秒単位)。

デフォルト: 120 秒

`Setenv`

(オプション) スクリプトに提供する環境変数のディクショナリ。これらの環境変数は、`Lifecycle.Setenv` で指定した変数を上書きします。

`Selections`

(オプション) このマニフェストに対して実行するグローバルライフサイクルのセクションを指定する選択キーのリスト。グローバルライフサイクルでは、任意のレベルで選択キーを使用してライフサイクルステップを定義し、ライフサイクルのサブセクションを選択することができます その後、コアデバイスは、このマニフェストの選択キーと一致するセクションを使用します。詳細については、「グローバルライフサイクルの例」を参照してください。

> **Important**
>
> コアデバイスは、このマニフェストがライフサイクルを指定していない場合にのみ、グローバルライフサイクルを使用します。

`all` 選択キーを指定して、選択キーを持たないグローバルライフサイクルのセクションを実行することができます。

`Artifacts`

(オプション) このマニフェストが定義するプラットフォーム上のコンポーネントのバイナリアーティファクトを定義するオブジェクトのリスト。例えば、コードまたはイメージをアーティファクトとして定義できます。

コンポーネントがデプロイされると、AWS IoT Greengrass Core ソフトウェアはコアデバイス上のフォルダにアーティファクトをダウンロードします。アーティファクトは、ソフトウェアがダウンロードした後に抽出するアーカイブファイルとして定義することもできます。

レシピ変数を使用すれば、コアデバイス上のアーティファクトがインストールされているフォルダへのパスを取得することができます。

- 通常のファイル - artifacts:path レシピ変数を使用して、アーティファクトが含まれるフォルダへのパスを取得します。例えば、レシピで `{artifacts:path}/my_script.py` を指定すれば、URI `s3://DOC-EXAMPLE-BUCKET/path/to/my_script.py` を持つアーティファクトへのパスを取得できます。
- 抽出されたアーカイブ - artifacts:decompressedPath レシピ変数を使用して、抽出されたアーカイブアーティファクトが含まれるフォルダへのパスを取得します。AWS IoT

Greengrass Core ソフトウェアは、各アーカイブをアーカイブと同じ名前のフォルダに抽出します。例えば、レシピで `{artifacts:decompressedPath}/my_archive/my_script.py` を指定すれば、URI `s3://DOC-EXAMPLE-BUCKET/path/to/my_archive.zip` を持つアーカイブアーティファクト内にある `my_script.py` へのパスを取得できます。

> **Note**
>
> ローカルコアデバイス上でアーカイブアーティファクトを持つコンポーネントを開発する場合、そのアーティファクトの URI がない可能性があります。アーティファクトを抽出する `Unarchive` オプションを使用してコンポーネントをテストするには、ファイル名がアーカイブアーティファクトファイルの名前と一致する URI を指定します。アーカイブアーティファクトのアップロード先となる予定の URI を指定することも、新しいプレースホルダ URI を指定することもできます。例えば、ローカルデプロイ時に `my_archive.zip` アーティファクトを抽出する場合には、`s3://DOC-EXAMPLE-BUCKET/my_archive.zip` を指定できます。

各オブジェクトには、次の情報が含まれます:

`URI`

S3 バケット内のアーティファクトの URI。AWS IoT Greengrass Core ソフトウェアは、コンポーネントのインストール時にこの URI からアーティファクトをフェッチしますが、アーティファクトがデバイスにすでに存在する場合はフェッチしません。各アーティファクトには、各マニフェスト内に一意のファイル名がある必要があります。

`Unarchive`

(オプション) 解凍するアーカイブのタイプ。次のオプションから選択します。

- `NONE` - ファイルは解凍するアーカイブではありません。AWS IoT Greengrass Core ソフトウェアによって、コアデバイス上のフォルダにアーティファクトがインストールされます。artifacts:path レシピ変数を使用して、このフォルダへのパスを取得することができます。
- `ZIP` - ファイルは ZIP アーカイブです。AWS IoT Greengrass Core ソフトウェアは、アーカイブをアーカイブと同じ名前のフォルダに抽出します。artifacts:decompressedPath レシピ変数を使用して、このフォルダが含まれるフォルダへのパスを取得することができます。

デフォルトは `NONE` です。

`Permission`

(オプション) このアーティファクトファイルに設定するアクセス許可を定義するオブジェクト。読み取り許可と実行許可を設定できます。

> **Note**
>
> 書き込み許可は設定できません。これは、AWS IoT Greengrass Core ソフトウェアが、アーティファクトフォルダ内のアーティファクトファイルをコンポーネントが編集することを許可していないからです。コンポーネント内のアーティファクトファイルを編集するには、別の場所にコピーするか、新しいアーティファクトファイルをパブリッシュしてデプロイします。

アーチファクトを解凍するアーカイブとして定義すると、AWS IoT Greengrass Core ソフトウェアは、アーカイブから解凍するファイルに対してこれらのアクセス許可を設定します。AWS IoT Greengrass Core ソフトウェアは、フォルダのアクセス権限を `Read` および `Execute` に対して `ALL` に設定します。これにより、コンポーネントはフォルダ内の解凍されたファイルを表示できるようになります。アーカイブからの個々のファイルにアクセス許可を設定する場合は、インストールライフサイクルスクリプトでアクセス許可を設定することができます。

このオブジェクトには、次の情報が含まれます。

`Read`

(オプション) このアーティファクトファイルに設定する読み取りアクセス許可。このコンポーネントに依存するコンポーネントなどの他のコンポーネントがこのアーティファクトにアクセスできるようにするには、`ALL` を指定します。次のオプションから選択します。

- `NONE` - ファイルは読み取れません。
- `OWNER` - このコンポーネントを実行するように設定したシステムユーザーがファイルを読み取ることができます。
- `ALL` - このファイルはすべてのユーザーが読み取ることができます。

デフォルトは `OWNER` です。

`Execute`

(オプション) このアーティファクトファイルに設定する実行アクセス許可。`Execute` アクセス許可は、`Read` アクセス許可を意味します。例えば、`Execute` に対して `ALL` を指定すると、すべてのユーザーがこのアーティファクトファイルを読み取り、実行できるようになります。

次のオプションから選択します。

- `NONE` - ファイルは実行できません。
- `OWNER` - このコンポーネントを実行するように設定したシステムユーザーがファイルを実行することができます。
- `ALL` - このファイルはすべてのユーザーが実行することができます。

デフォルトは `NONE` です。

`Digest`

(読み取り専用) アーティファクトの暗号化ダイジェストハッシュ。コンポーネントを作成するときには、AWS IoT Greengrass はハッシュアルゴリズムを使用して、アーティファクトファイルのハッシュを計算します。その後、コンポーネントをデプロイするときに、Greengrass nucleus がダウンロードされたアーティファクトのハッシュを計算し、このダイジェストとハッシュを比較することでインストール前にアーティファクトを検証します。ハッシュがダイジェストと一致しない場合、デプロイは失敗します。

このパラメータを設定すると、AWS IoT Greengrass はコンポーネントの作成時に設定した値を置き換えます。

`Algorithm`

(読み取り専用) アーティファクトのダイジェストハッシュを計算するために AWS IoT Greengrass が使用するハッシュアルゴリズム。

このパラメータを設定すると、AWS IoT Greengrass はコンポーネントの作成時に設定した値を置き換えます。

`Lifecycle`

コンポーネントをインストールして実行する方法を定義するオブジェクト。コアデバイスは、使用するマニフェストでライフサイクルが指定されていない場合にのみ、グローバルライフサイクルを使用します。

> **Note**
>
> このライフサイクルはマニフェスト外で定義します。マニフェストと一致するプラットフォームに適用されるマニフェストのライフサイクルを定義することもできます。

グローバルライフサイクルでは、各マニフェストで指定した特定の選択キーに対して実行するライフサイクルを指定することができます。選択キーは、各マニフェストに対して実行するグローバルライフサイクルのセクションを識別する文字列です。

選択キーがないセクションのデフォルトは、all 選択キーです。つまり、マニフェストで all 選択キーを指定することで、選択キーがないグローバルライフサイクルのセクションを実行することができることになります。グローバルライフサイクルで all 選択キーを指定する必要はありません

マニフェストでライフサイクルまたは選択キーが定義されていない場合、コアデバイスはデフォルトで all 選択を使用します。つまりこの場合、コアデバイスは、選択キーを使用しないグローバルライフサイクルのセクションを使用することになります。

このオブジェクトには、マニフェストのライフサイクルと同じ情報が含まれていますが、任意のレベルで選択キーを指定することで、ライフサイクルのサブセクションを選択することができます。

> **Tip**
>
> 選択キーとライフサイクルキーが競合することがないように、各選択キーには小文字のみを使用することを推奨します。ライフサイクルキーは大文字で始まります。

Example トップレベルの選択キーを使用したグローバルライフサイクルの例

```
Lifecycle:
  key1:
    install:
      Skipif: either onpath executable or exists file
      Script: command1
  key2:
    install:
      Script: command2
```

```
  all:
    install:
      Script: command3
```

Example 末尾レベルの選択キーを使用したグローバルライフサイクルの例

```
Lifecycle:
  install:
    Script:
      key1: command1
      key2: command2
      all: command3
```

Example 複数のレベルの選択キーを使用したグローバルライフサイクルの例

```
Lifecycle:
  key1:
    install:
      Skipif: either onpath executable or exists file
      Script: command1
  key2:
    install:
      Script: command2
  all:
    install:
      Script:
        key3: command3
        key4: command4
        all: command5
```

レシピ変数

レシピ変数は、現在のコンポーネントと nucleus の情報を公開し、レシピで使用できるようにします。例えば、レシピ変数を使用して、ライフサイクルスクリプトで実行するアプリケーションにコンポーネント設定パラメータを渡すことができます。

レシピ変数は、以下のコンポーネントレシピの各セクションで使用できます。

- ライフサイクル定義。

- Greengrass nucleus v2.6.0 以降を使用し、 設定オプションを に設定した場合のコンポーネントinterpolateComponentConfiguration設定定義true。レシピ変数は、コンポーネント設定の更新をデプロイする際にも、使用することが可能です。

レシピ変数は {recipe_variable} 構文を使用します。中括弧はレシピ変数を示しています。

AWS IoT Greengrass は次のレシピ変数をサポートしています。

component_dependency_name:configuration:*json_pointer*

このレシピが定義するコンポーネント、またはこのコンポーネントが依存するコンポーネントの設定パラメータの値。

この変数を使用して、コンポーネントのライフサイクルで実行するスクリプトにパラメータを提供することができます。

> **Note**
>
> AWS IoT Greengrass は、コンポーネントのライフサイクル定義でのみ、このレシピ変数をサポートします。

このレシピ変数には、次の入力があります。

- component_dependency_name - (オプション) クエリを実行するコンポーネント依存関係の名前。このレシピが定義するコンポーネントを照会する場合は、このセグメントを省略します。直接的な依存関係のみを指定できます。
- json_pointer - 評価する設定値への JSON ポインタ。JSON ポインタはフォワードスラッシュ (/) で始まります。ネストされたコンポーネント設定の値を識別するには、フォワードスラッシュ (/) を使用して、設定の各レベルのキーを区切ります。数字をキーとして使用して、リスト内のインデックスを指定できます。詳細については、「JSON ポインタの仕様」を参照してください。

  AWS IoT Greengrass Core は YAML 形式のレシピに JSON ポインタを使用します。

JSON ポインタは次のノードタイプを参照できます。

- 値ノード。AWS IoT GreengrassCore は、レシピ変数を値の文字列表現に置き換えます。Null 値は文字列の null に置き換えられます。

- オブジェクトノード。AWS IoT GreengrassCore は、レシピ変数をそのオブジェクトのシリアル化された JSON 文字列表現に置き換えます。
- ノードなし。AWS IoT GreengrassCore はレシピ変数を置き換えません。

例えば、`{configuration:/Message}` レシピ変数は、コンポーネント設定の `Message` キーの値を取得します。`{com.example.MyComponentDependency:configuration:/server/port}` レシピ変数は、コンポーネント依存関係の `server` 設定オブジェクトにある `port` の値を取得します。

component_dependency_name`:artifacts:path`

このレシピが定義するコンポーネント、またはこのコンポーネントが依存するコンポーネントのアーティファクトのルートパス。

コンポーネントがインストールされると、AWS IoT Greengrass はコンポーネントのアーティファクトを、この変数が公開するフォルダにコピーします。この変数を使用して、コンポーネントのライフサイクルで実行するスクリプトの場所などを特定できます。

このパスのフォルダは読み取り専用です。アーティファクトファイルを変更するには、ファイルを現在の作業ディレクトリなどの別の場所にコピーします (`$PWD` または `.`)。その後、その場所でファイルを変更します。

コンポーネントの依存関係からアーティファクトを読み取るか実行するには、そのアーティファクトの `Read` または `Execute` のアクセス許可が `ALL` である必要があります。詳細については、コンポーネントレシピで定義した「アーティファクトアクセス許可」を参照してください。

このレシピ変数には、次の入力があります。

- component_dependency_name - (オプション) クエリを実行するコンポーネント依存関係の名前。このレシピが定義するコンポーネントを照会する場合は、このセグメントを省略します。直接的な依存関係のみを指定できます。

component_dependency_name`:artifacts:decompressedPath`

このレシピが定義するコンポーネント、またはこのコンポーネントが依存するコンポーネントの解凍されたアーカイブアーティファクトのルートパス。

コンポーネントがインストールされると、AWS IoT Greengrass はコンポーネントのアーカイブアーティファクトを、この変数が公開するフォルダにコピーします。この変数を使用して、コンポーネントのライフサイクルで実行するスクリプトの場所などを特定できます。

各アーティファクトは、解凍されたパス内のフォルダに解凍されます。フォルダの名前はアーティファクトと同じ名前から拡張子を引いたものになります。例えば、`models.zip` という名前

の ZIP アーティファクトは、`{artifacts:decompressedPath}/models` フォルダに解凍されます。

このパスのフォルダは読み取り専用です。アーティファクトファイルを変更するには、ファイルを現在の作業ディレクトリなどの別の場所にコピーします (`$PWD` または `.`)。その後、その場所でファイルを変更します。

コンポーネントの依存関係からアーティファクトを読み取るか実行するには、そのアーティファクトの `Read` または `Execute` のアクセス許可が `ALL` である必要があります。詳細については、コンポーネントレシピで定義した「アーティファクトアクセス許可」を参照してください。

このレシピ変数には、次の入力があります。

- `component_dependency_name` - (オプション) クエリを実行するコンポーネント依存関係の名前。このレシピが定義するコンポーネントを照会する場合は、このセグメントを省略します。直接的な依存関係のみを指定できます。

`component_dependency_name``:work:path`

この機能は、Greengrass nucleus コンポーネントの v2.0.4 以降に利用できます。

このレシピが定義するコンポーネント、またはこのコンポーネントが依存するコンポーネントのワークパス。このレシピ変数の値は、コンポーネントのコンテキストから実行されたときの `$PWD` 環境変数と pwd コマンドの出力と同じです。

このレシピ変数を使用して、コンポーネントと依存関係間でファイルを共有できます。

このパスのフォルダは、このレシピが定義するコンポーネントと、同じユーザーおよびグループとして実行される他のコンポーネントが、読み取り書き込むことができます。

このレシピ変数には、次の入力があります。

- `component_dependency_name` - (オプション) クエリを実行するコンポーネント依存関係の名前。このレシピが定義するコンポーネントを照会する場合は、このセグメントを省略します。直接的な依存関係のみを指定できます。

`kernel:rootPath`

AWS IoT Greengrass Core のルートパス。

`iot:thingName`

この機能は、Greengrass nucleus コンポーネントの v2.3.0 以降に利用できます。

コアデバイスの AWS IoT モノの名前。

レシピの例

コンポーネントのレシピを作成する際には、次のレシピ例を参照してください。

AWS IoT Greengrass は Greengrass ソフトウェアカタログと呼ばれる Greengrass コンポーネントのインデックスをキュレーションします。このカタログは、Greengrass コミュニティによって開発された Greengrass コンポーネントを追跡します。このカタログから、コンポーネントをダウンロード、変更、デプロイして Greengrass アプリケーションを作成できます。詳細については、「コミュニティコンポーネント」を参照してください。

トピック

- Hello World コンポーネントレシピ
- Python ランタイムコンポーネントの例
- 複数のフィールドを指定するコンポーネントレシピ

Hello World コンポーネントレシピ

次のレシピは、Python スクリプトを実行する Hello World コンポーネントについて説明しています。このコンポーネントはすべてのプラットフォームをサポートしており、AWS IoT Greengrass が Python スクリプトに引数としてわたす `Message` パラメータを受け入れます。これは、入門チュートリアルの Hello World コンポーネントのためのレシピです。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.HelloWorld",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "My first AWS IoT Greengrass component.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "world"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
```

```
      "Lifecycle": {
        "run": "python3 -u {artifacts:path}/hello_world.py {configuration:/Message}"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "run": "py -3 -u {artifacts:path}/hello_world.py {configuration:/Message}"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: My first AWS IoT Greengrass component.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    Message: world
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      run: |
        python3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
  - Platform:
      os: windows
    Lifecycle:
      run: |
        py -3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
```

Python ランタイムコンポーネントの例

次のレシピは、Python をインストールするコンポーネントについて説明しています。このコンポーネントは 64 ビット Linux デバイスをサポートします。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PythonRuntime",
  "ComponentDescription": "Installs Python 3.7",
  "ComponentPublisher": "Amazon",
  "ComponentVersion": "3.7.0",
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "architecture": "amd64"
      },
      "Lifecycle": {
        "install": "apt-get update\napt-get install python3.7"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PythonRuntime
ComponentDescription: Installs Python 3.7
ComponentPublisher: Amazon
ComponentVersion: '3.7.0'
Manifests:
  - Platform:
      os: linux
      architecture: amd64
    Lifecycle:
      install: |
        apt-get update
        apt-get install python3.7
```

複数のフィールドを指定するコンポーネントレシピ

次のコンポーネントレシピでは、複数のレシピフィールドを使用します。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.FooService",
  "ComponentDescription": "Complete recipe for AWS IoT Greengrass components",
  "ComponentPublisher": "Amazon",
  "ComponentVersion": "1.0.0",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "TestParam": "TestValue"
    }
  },
  "ComponentDependencies": {
    "BarService": {
      "VersionRequirement": "^1.1.0",
      "DependencyType": "SOFT"
    },
    "BazService": {
      "VersionRequirement": "^2.0.0"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "architecture": "amd64"
      },
      "Lifecycle": {
        "install": {
          "Skipif": "onpath git",
          "Script": "sudo apt-get install git"
        },
        "Setenv": {
          "environment_variable1": "variable_value1",
          "environment_variable2": "variable_value2"
        }
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/hello_world.zip",
          "Unarchive": "ZIP"
        },
        {
```

```
          "URI": "s3://DOC-EXAMPLE-BUCKET/hello_world_linux.py"
        }
      ]
    },
    {
      "Lifecycle": {
        "install": {
          "Skipif": "onpath git",
          "Script": "sudo apt-get install git",
          "RequiresPrivilege": "true"
        }
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/hello_world.py"
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.FooService
ComponentDescription: Complete recipe for AWS IoT Greengrass components
ComponentPublisher: Amazon
ComponentVersion: 1.0.0
ComponentConfiguration:
  DefaultConfiguration:
    TestParam: TestValue
ComponentDependencies:
  BarService:
    VersionRequirement: ^1.1.0
    DependencyType: SOFT
  BazService:
    VersionRequirement: ^2.0.0
Manifests:
  - Platform:
      os: linux
      architecture: amd64
    Lifecycle:
```

```
      install:
        Skipif: onpath git
        Script: sudo apt-get install git
      Setenv:
        environment_variable1: variable_value1
        environment_variable2: variable_value2
    Artifacts:
      - URI: 's3://DOC-EXAMPLE-BUCKET/hello_world.zip'
        Unarchive: ZIP
      - URI: 's3://DOC-EXAMPLE-BUCKET/hello_world_linux.py'
  - Lifecycle:
      install:
        Skipif: onpath git
        Script: sudo apt-get install git
        RequiresPrivilege: 'true'
    Artifacts:
      - URI: 's3://DOC-EXAMPLE-BUCKET/hello_world.py'
```

コンポーネントの環境変数リファレンス

AWS IoT Greengrass Core ソフトウェアは、コンポーネントのライフサイクルスクリプトを実行するときに、環境変数を設定します。これらのコンポーネント内の環境変数を取得することで、モノの名前、AWS リージョン、および Greengrass nucleus バージョンを取得することができます。また、このソフトウェアは、コンポーネントがプロセス間通信 SDK を使用し、AWS サービスとやり取りするために必要な環境変数も設定します。

コンポーネントのライフサイクルスクリプトにカスタム環境変数を設定することもできます。詳細については「Setenv」を参照してください。

AWS IoT Greengrass Core ソフトウェアは、以下の環境変数を設定します。

`AWS_IOT_THING_NAME`

この Greengrass コアデバイスを表す AWS IoT モノの名前。

`AWS_REGION`

この Greengrass コアデバイスが動作する場所となる AWS リージョン。

AWS SDK はこの環境変数を使用して、使用するデフォルトのリージョンを識別します。この変数は `AWS_DEFAULT_REGION` と同等です。

`AWS_DEFAULT_REGION`

この Greengrass コアデバイスが動作する場所となる AWS リージョン。

AWS CLI はこの環境変数を使用して、使用するデフォルトのリージョンを識別します。この変数は `AWS_REGION` と同等です。

`GGC_VERSION`

この Greengrass コアデバイスで実行される Greengrass nucleus コンポーネントのバージョン。

`GG_ROOT_CA_PATH`

この機能は、Greengrass nucleus コンポーネントの v2.5.5 以降に利用できます。

Greengrass nucleus が使用するルート認証局 (CA) 証明書へのパス。

`AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT`

コンポーネントが、AWS IoT Greengrass Core ソフトウェアとの通信に使用する IPC ソケットへのパス。詳細については、「AWS IoT Device SDK を使用して Greengrass nucleus、その他のコンポーネント、および AWS IoT Core と通信する」を参照してください。

`SVCUID`

コンポーネントが IPC ソケットへの接続と、AWS IoT Greengrass Core ソフトウェアとの通信のために使用するシークレットトークン。詳細については、「AWS IoT Device SDK を使用して Greengrass nucleus、その他のコンポーネント、および AWS IoT Core と通信する」を参照してください。

`AWS_CONTAINER_AUTHORIZATION_TOKEN`

コンポーネントが、トークン交換サービスコンポーネントから認証情報を取得するために使用するシークレットトークン。

`AWS_CONTAINER_CREDENTIALS_FULL_URI`

コンポーネントが、トークン交換サービスコンポーネントから認証情報を取得するためにリクエストする URI。

デバイスに AWS IoT Greengrass コンポーネントのデプロイ

デバイスまたはデバイスのグループにコンポーネントを配備するために AWS IoT Greengrass を使用できます。デプロイを使用して、デバイスに送信するコンポーネントとコンフィギュレーションを

定義します。AWS IoT Greengrass は、Greengrass のコアデバイスを表すターゲット、AWS IoT モノ、またはモノのグループにデプロイします。AWS IoT Core ジョブを使用してコアデバイスにデプロイします。ジョブのデバイスへの展開方法を設定することができます。

コアデバイスのデプロイ

各コアデバイスは、そのデバイスのためのデプロイのコンポーネントを実行します。同じターゲットへの新しいデプロイは、ターゲットへの以前のデプロイ置を上書きします。デプロイを作成するとき、コアデバイスの既存のソフトウェアに適用するコンポーネントと設定を定義します。

ターゲットのデプロイを改訂するとき、前の改訂のコンポーネントを新しい改訂のコンポーネントに置き換えます。例えば、コンポーネント ログマネージャー と シークレットマネージャー をモノグループ `TestGroup` にデプロイします。次に、シークレットマネージャーのコンポーネントのみを指定する `TestGroup` 別のデプロイを作成します。その結果、そのグループのコアデバイスは、ログマネージャーを実行しなくなりました。

プラットフォーム依存の解決

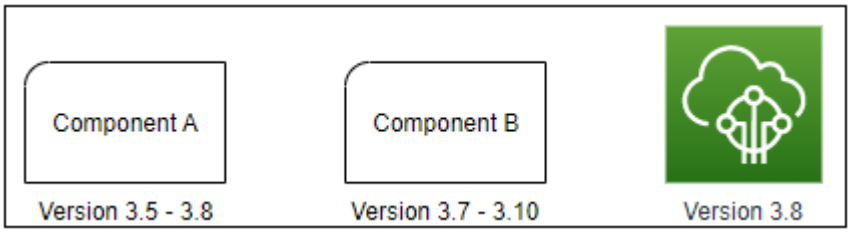
コアデバイスはデプロイを受け取ると、そのコンポーネントがコアデバイスと互換性があるかどうかを確認します。例えば、Windows のターゲットに Firehose をデプロイした場合、デプロイは失敗します。

コンポーネント依存の解決

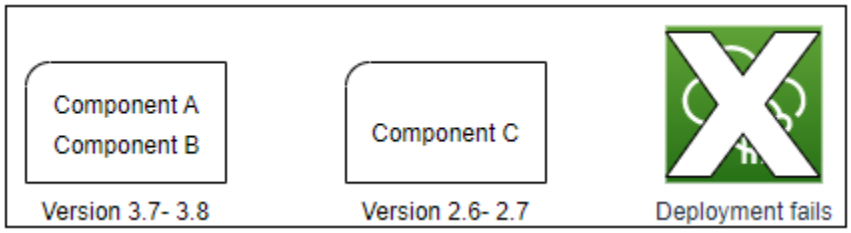
また、コアデバイスは、各コンポーネントの依存関係が、このモノグループに他のコンポーネントをデプロイする際のバージョン制約に適合しているかどうかをチェックします。コンポーネントのバージョン制約が重複する場合、Greengrass はコンポーネントの適用可能な最大バージョンを使用します。例:

- コンポーネント A を `TestGroup` にデプロイします。コンポーネント A は `com.example.PythonRuntime` コンポーネントバージョン 3.5 ~ 3.10 に依存します。
- 次に、コンポーネント B を `TestGroup` にデプロイします。コンポーネント B は `com.example.PythonRuntime` コンポーネントバージョン 3.7 ~ 3.8 に依存します。

この場合は結果的に、`TestGroup` のコアデバイスでは、バージョン 3.8 の `com.example.PythonRuntime` コンポーネントの導入が可能と判断されます。重複したバージョン制約のうち、このバージョン番号が最も高いためです。



次に、コンポーネント C を TestGroup にデプロイします。コンポーネント C は com.example.PythonRuntime コンポーネントバージョン 2.6 ~ 2.7 に依存します。制約条件である 2.6 ~ 2.7 および 3.7 ~ 3.8 を満たすコンポーネントのバージョンがないため、このデプロイは失敗します。



モノのグループからデバイスを削除する

モノグループからコアデバイスを削除するとき、コンポーネントのデプロイ動作は、コアデバイスが実行する Greengrass nucleus のバージョンによって異なります。

2.5.1 and later

コアデバイスをモノグループから削除する場合、AWS IoT ポリシーが greengrass:ListThingGroupsForCoreDevice 許可を与えているかどうかで、動作が異なります。コアデバイスの許可と AWS IoT ポリシーの詳細については、「AWS IoT Greengrass のデバイス認証と認可」を参照してください。

- AWS IoT ポリシーはこの許可を付与する場合

  モノグループからコアデバイスを削除すると、デバイスへの次回のデプロイ時に AWS IoT Greengrass によってモノグループのコンポーネントが削除されます。デバイス上のコンポーネントが次のデプロイに含まれる場合、そのコンポーネントはデバイスから削除されません。

- AWS IoT ポリシーがこの許可を付与しない場合

  モノグループからコアデバイスを削除するとき、AWS IoT Greengrass はデバイスからそのモノグループのコンポーネントを削除しません。

デバイスからコンポーネントを削除するには、Greengrass CLI の [デプロイ作成] コマンドを使用します。削除するコンポーネントを `--remove` 引数で指定し、`--groupId` 引数でモノグループを指定します。

2.5.0

モノグループからコアデバイスを削除すると、デバイスへの次回のデプロイ時に AWS IoT Greengrass によってモノグループのコンポーネントが削除されます。デバイス上のコンポーネントが次のデプロイに含まれる場合、そのコンポーネントはデバイスから削除されません。

この動作は、コアデバイスの AWS IoT ポリシーが `greengrass:ListThingGroupsForCoreDevice` 許可を付与することを要件とします。コアデバイスにこの許可がない場合、コアデバイスはデプロイの適用に失敗します。詳細については、「AWS IoT Greengrass のデバイス認証と認可」を参照してください。

2.0.x - 2.4.x

モノグループからコアデバイスを削除するとき、AWS IoT Greengrass はデバイスからそのモノグループのコンポーネントを削除しません。

デバイスからコンポーネントを削除するには、Greengrass CLI の [デプロイ作成] コマンドを使用します。削除するコンポーネントを `--remove` 引数で指定し、`--groupId` 引数でモノグループを指定します。

デプロイ

デプロイは継続的です。デプロイを作成するとき、AWS IoT Greengrass はオンラインのターゲットデバイスにデプロイをロールアウトします。ターゲットデバイスがオンラインではない場合、次回 AWS IoT Greengrass に接続したときにデプロイを受信します。ターゲットモノグループにコアデバイスを追加するとき、AWS IoT Greengrass は、デバイスに対してそのモノグループの最新デプロイを送信します。

コアデバイスがコンポーネントをデプロイする前に、デフォルトではデバイス上の各コンポーネントに通知します。Greengrass コンポーネントは、通知に応答してデプロイを延期できます。デバイスのバッテリ残量が少ない場合や、中断できないプロセスを実行している場合などに、デプロイを延期できます。詳細については、「チュートリアル: コンポーネントの更新を延期する Greengrass コンポーネントを開発する」を参照してください。デプロイを作成するときに、コンポーネントに通知せずにデプロイするように設定することができます。

各ターゲットのモノまたはモノグループは、一度に 1 件のデプロイを持つことができます。これは、ターゲットのデプロイを作成するとき、AWS IoT Greengrass はそのターゲットのデプロイにおける前の改訂をデプロイしなくなります。

デプロイオプション

デプロイは、更新を受信するデバイスと更新のデプロイ方法の制御を可能にするいくつかのオプションを提供します。デプロイを作成するとき、次のオプションを設定できます。

- AWS IoT Greengrass コンポーネント

  ターゲットデバイスにインストールして実行するコンポーネントを定義します。Greengrass コアデバイスにデプロイして実行する AWS IoT Greengrass コンポーネントとソフトウェアモジュールです。コンポーネントがデバイスのプラットフォームをサポートしている場合のみ、デバイスがコンポーネントを受信します。これにより、ターゲットデバイスが複数のプラットフォームで実行されている場合でも、デバイスのグループにデプロイできます。コンポーネントがデバイスのプラットフォームをサポートしていない場合、コンポーネントはデバイスにデプロイしません。

  カスタムコンポーネントと AWS が提供するコンポーネントをデバイスにデプロイできます。コンポーネントをデプロイすると、AWS IoT Greengrass はコンポーネントの従属関係を特定し、それらもデプロイします。詳細については、「AWS IoT Greengrass コンポーネントを開発する と AWS が提供したコンポーネント」を参照してください。

  各コンポーネントにデプロイするバージョンと設定の更新を定義します。設定更新は、コアデバイスにコンポーネントが存在しない場合、コアデバイスのコンポーネントの既存設定またはコンポーネントのデフォルト設定を修正する方法を指定します。デフォルト値にリセットする設定値と、コアデバイスにマージする新しい設定値を指定することができます。コアデバイスが異なるターゲット用のデプロイを受信し、かつ各デプロイが互換性のあるコンポーネントバージョンを指定するとき、コアデバイスは、デプロイを作成したときのタイムスタンプに基づいて、設定更新を順序に従って適用します。詳細については、「コンポーネント設定の更新」を参照してください。

  > **Important**
  >
  > コンポーネントをデプロイする際、AWS IoT Greengrass は、そのコンポーネントの従属関係においてサポートされた最新のバージョンをインストールします。このため、新しいデバイスをモノグループに追加したり、これらのデバイスを対象とするデプロイを更新すると、AWS が提供するパブリックコンポーネントの新しいパッチバージョンが自動的にコアデバイスにデプロイされる場合があります。nucleus の更新など、一部の自動更新により、デバイスに予期せぬ再起動が発生することがあります。

デバイスで実行されているコンポーネントに不要に更新されることを防ぐには、デプロイを作成する際、そのコンポーネントの優先バージョンを直接含めることをお勧めします。AWS IoT Greengrass Core ソフトウェアの更新動作の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

- デプロイポリシー

  設定を安全にデプロイできるタイミングと、デプロイが失敗した場合の対処方法を定義します。コンポーネントが更新できることを報告することを待機するかどうか指定できます。失敗するデプロイを適用した場合、デバイスを以の設定にロールバックするかどうか指定することもできます。

- 設定を停止

  デプロイを停止するタイミングと方法を定義します。定義した基準が満たされると、デプロイは停止して失敗します。例えば、最小数のデバイスがデプロイを受信した後、そのデプロイの適用に失敗したデバイスがある割合に達した場合、デプロイを停止するように設定できます。

- ロールアウト設定

  デプロイがターゲットデバイスにロールアウトされるレートを定義します。最小レートと最大レートの閾値で指数関数的レート増加を設定できます。

- タイムアウト設定

  各デバイスがデプロイに適用する必要がある最大時間を定義します。デバイスが指定した時間を超えると、デバイスはデプロイの適用に失敗します。

Important

カスタムコンポーネントは S3 バケットにアーティファクトを定義できます。AWS IoT Greengrass Core ソフトウェアがコンポーネントをデプロイするとき、コンポーネントのアーティファクトを AWS クラウド からダウンロードします。コアデバイスのロールは、デフォルトで S3 バケットへのアクセスを許可しません。S3 バケットのアーティファクトを定義するカスタムコンポーネントをデプロイするには、コアデバイスロールはそのバケットからアーティファクトをダウンロードする許可を付与する必要があります。詳細については、「コンポーネントのアーティファクトの S3 バケットへのアクセスを許可する」を参照してください。

トピック

デプロイの作成

モノまたはモノグループをターゲットとするデプロイを作成できます。

デプロイを作成するときは、デプロイするソフトウェアコンポーネントと、デプロイジョブをターゲットデバイスにロールアウトする方法を設定します。デプロイは、AWS CLI に提供する JSON ファイルで定義できます。

デプロイターゲットによって、コンポーネントを実行するデバイスが決まります。1 つのコアデバイスにデプロイするには、モノを指定します。複数のコアデバイスにデプロイするには、これらのデバイスが含まれるモノグループを指定します。モノのグループを設定する方法についての詳細は、「AWS IoT デベロッパーガイド」の「静的モノグループ」と「動的モノグループ」を参照してください。

このセクションのステップに従って、ターゲットへのデプロイを作成します。デプロイがあるターゲット上で、ソフトウェアコンポーネントを更新する方法についての詳細は、「展開の改訂」を参照してください。

> **Warning**
>
> CreateDeployment オペレーションでは、コアデバイスからコンポーネントをアンインストールできます。コンポーネントが新しいデプロイではなく以前のデプロイに存在する場合、コアデバイスはそのコンポーネントをアンインストールします。コンポーネントのアンインストールを回避するには、まず ListDeploymentsオペレーションを使用して、デプロイのターゲットに既存のデプロイがすでにあるかどうかを確認します。次に、新しいデプロイを作成するときに、 GetDeploymentオペレーションを使用して既存のデプロイから開始します。

デプロイを作成するには (AWS CLI)

1. deployment.json という名前のファイルを作成して、次の JSON オブジェクトをファイルにコピーします。*targetArn* をデプロイがターゲットとする AWS IoT モノまたはモノグループの ARN に置き換えます。モノおよびモノグループの ARN の形式は、次のとおりです。

   - モノ: `arn:aws:iot:region:account-id:thing/thingName`
   - モノのグループ: `arn:aws:iot:region:account-id:thinggroup/thingGroupName`

   ```
   {
     "targetArn": "targetArn"
   }
   ```

2. デプロイメント対象に、修正が必要な既存のデプロイメントがあるかどうかをチェックします。以下を行います:

   a. 次のコマンドを実行して、デプロイターゲットのデプロイを一覧表示します。*targetArn* をターゲット AWS IoT モノまたはモノグループの ARN に置き換えます。

   ```
   aws greengrassv2 list-deployments --target-arn targetArn
   ```

   レスポンスには、ターゲットの最新デプロイのリストが含まれています。レスポンスが空の場合は、ターゲットに既存のデプロイがありません。Step 3 はスキップできます。そうでない場合は、次のステップで使用するため、レスポンスから deploymentId をコピーします。

   > **Note**
   >
   > ターゲットの最新リビジョン以外のデプロイを修正することもできます。--history-filter ALL 引数を指定して、ターゲットのすべてのデプロイを一覧表示します。次に、修正するデプロイの ID をコピーします。

   b. 次のコマンドを実行して、デプロイの詳細を取得します。これらの詳細には、メタデータ、コンポーネント、ジョブ設定が含まれます。*deploymentId* を前のステップの ID に置き換えます。

   ```
   aws greengrassv2 get-deployment --deployment-id deploymentId
   ```

レスポンスには、デプロイの詳細が含まれています。

c. 前のコマンドのレスポンスにある次のキーと値のペアを deployment.json にコピーします。これらの値を、新しいデプロイに変更することができます。

- deploymentName - デプロイの名前。
- components - デプロイのコンポーネント。コンポーネントをアンインストールする場合は、このオブジェクトから削除してください。
- deploymentPolicies - デプロイのポリシー。
- iotJobConfiguration - デプロイのジョブ設定。
- tags - デプロイのタグ。

3. (オプション) デプロイの名前を定義します。*deploymentName* をデプロイの名前で置き換えます。

```
{
  "targetArn": "targetArn",
  "deploymentName": "deploymentName"
}
```

4. 各コンポーネントを追加して、ターゲットデバイスをデプロイします。そのためには、キーと値のペアを components オブジェクトに追加します。ここでのキーはコンポーネント名、値はそのコンポーネントの詳細が含まれるオブジェクトになります。追加する各コンポーネントに対して、次の詳細を指定します。

- version - デプロイするコンポーネントのバージョン。
- configurationUpdate - デプロイする設定の更新。更新とは、各ターゲットデバイス上のコンポーネントにある既存の設定、またはターゲットデバイス上に既存の設定がない場合はコンポーネントのデフォルト設定を変更するパッチ操作のことを指します。次の設定更新を指定できます。
  - 更新のリセット (reset) - (オプション) ターゲットデバイスのデフォルト値にリセットする設定値を定義する JSON ポインターのリスト。AWS IoT Greengrass Core ソフトウェアは、マージ更新を適用する前にリセット更新を適用します。詳細については、「更新のリセット」を参照してください。
  - マージ更新 (merge) - (オプション) ターゲットデバイスにマージする設定値を定義する JSON ドキュメント。JSON ドキュメントは文字列としてシリアル化する必要があります。詳細については、「マージの更新」を参照してください。

- `runWith` - (オプション) AWS IoT Greengrass Core ソフトウェアが、コアデバイス上でこのコンポーネントのプロセスを実行するために使用するシステムプロセスオプション。`runWith` オブジェクト内のパラメータを省略した場合、AWS IoT Greengrass Core ソフトウェアは Greengrass nucleus コンポーネントで設定したデフォルト値を使用します。

  以下のいずれかのオプションを指定できます。

  - `posixUser` - Linux コアデバイスでこのコンポーネントを実行する際に使用する POSIX システムユーザーおよびオプションでグループ。ユーザーまたはグループを指定する場合は、各 Linux コアデバイス上に存在している必要があります。ユーザーとグループを `user:group` の形式に従ってコロン (`:`) で区切って指定します。グループはオプションです。グループを指定しなかった場合、AWS IoT Greengrass Core ソフトウェアは、ユーザーのプライマリグループを使用します。詳細については、「コンポーネントを実行するユーザーを設定する」を参照してください。
  - `windowsUser` – Windows コアデバイスでこのコンポーネントを実行する際に使用する Windows ユーザー。ユーザーは各 Windows コアデバイスに存在し、その名前とパスワードは LocalSystem アカウントの認証情報マネージャーインスタンスに保存されている必要があります。詳細については、「コンポーネントを実行するユーザーを設定する」を参照してください。

    この機能は、Greengrass nucleus コンポーネントの v2.5.0 以降に利用できます。
  - `systemResourceLimits` - このコンポーネントのプロセスに適用されるシステムリソースの制限。システムリソースの制限を、ジェネリックおよびコンテナ化されていない Lambda コンポーネントプロセスに適用することができます。詳細については、「コンポーネントのシステムリソース制限を設定する」を参照してください。

    以下のいずれかのオプションを指定できます。

    - `cpus` – このコンポーネントのプロセスがコアデバイスで使用できる CPU 時間の最大量。コアデバイスの合計 CPU 時間は、デバイスの CPU コア数と同じです。例えば、4 つの CPU コアを持つコアデバイスの場合は、この値を `2` に設定することで、このコンポーネントのプロセスを各 CPU コアの 50% の使用率に制限することができます。CPU コアが 1 つのデバイスの場合は、この値を `0.25` に設定することで、このコンポーネントのプロセスを CPU の 25% の使用率に制限することができます。この値を CPU コア数よりも大きい値に設定すると、AWS IoT Greengrass Core ソフトウェアは、コンポーネントの CPU 使用率に制限をかけません。
    - `memory` – このコンポーネントのプロセスがコアデバイスで使用できる RAM の最大量 (キロバイト単位)。

この機能は、Greengrass nucleus コンポーネントの v2.4.0 以降に利用できます。AWS IoT Greengrass は、現在 Windows コアデバイスにこの機能をサポートしていません。

Example 基本設定更新の例

次の components オブジェクト例は、pythonVersion という名前の設定パラメータを必要とするコンポーネントである com.example.PythonRuntime をデプロイするように指定しています。

```
{
  "targetArn": "targetArn",
  "deploymentName": "deploymentName",
  "components": {
    "com.example.PythonRuntime": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "merge": "{\"pythonVersion\":\"3.7\"}"
      }
    }
  }
}
```

Example リセット更新とマージ更新による設定更新の例

com.example.IndustrialDashboard は産業用ダッシュボードコンポーネントの一例です。次のデフォルト設定が設定されています。

```
{
  "name": null,
  "mode": "REQUEST",
  "network": {
    "useHttps": true,
    "port": {
      "http": 80,
      "https": 443
    },
  },
  "tags": []
```

```
}
```

次の設定更新では、次の手順が指定されています。

1. HTTPS 設定をデフォルト値 (true) にリセットする。
2. 産業タグのリストを空のリストにリセットする。
3. 2 つのボイラーの温度と圧力のデータストリームを識別する産業用タグのリストをマージする。

```
{
  "reset": [
    "/network/useHttps",
    "/tags"
  ],
  "merge": {
    "tags": [
      "/boiler/1/temperature",
      "/boiler/1/pressure",
      "/boiler/2/temperature",
      "/boiler/2/pressure"
    ]
  }
}
```

以下の components オブジェクト例では、この産業用ダッシュボードコンポーネントと設定更新をデプロイするように指定しています。

```
{
  "targetArn": "targetArn",
  "deploymentName": "deploymentName",
  "components": {
    "com.example.IndustrialDashboard": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "reset": [
          "/network/useHttps",
          "/tags"
        ],
        "merge": "{\"tags\":[\"/boiler/1/temperature\",\"/boiler/1/pressure\",\"/
boiler/2/temperature\",\"/boiler/2/pressure\"]}"
```

```
      }
    }
  }
}
```

5. (オプション) デプロイのデプロイポリシーを定義します。コアデバイスがデプロイを安全に適用できるタイミングや、コアデバイスがデプロイの適用に失敗した場合の対処方法を設定できます。そのためには、deploymentPolicies オブジェクトを deployment.json に追加し、以下のいずれかを実行します。

   1. (オプション) コンポーネント更新ポリシー (componentUpdatePolicy) を指定します。このポリシーは、コンポーネントをデプロイする準備が整うまで、コンポーネントの更新を延期できるかどうかを定義します。例えば、再起動して更新を適用する前に、リソースをクリーンアップしたり、重要なアクションを完了する必要がある場合などです。このポリシーは、コンポーネントが更新通知に対してレスポンスする必要がある時間枠も定義します。

      このポリシーは、以下のパラメータを使用するオブジェクトです。

      - action - (オプション) コンポーネントに通知し、更新の準備が整ったときに報告されるまで待機するかどうかを示します。次のオプションから選択します。
        - NOTIFY_COMPONENTS - デプロイは、コンポーネントを停止して更新する前に、各コンポーネントに通知します。コンポーネントは SubscribeToComponentUpdates IPC 操作を使用して、これらの通知を受信することができます。
        - SKIP_NOTIFY_COMPONENTS - デプロイは、コンポーネントに通知せず、安全に更新できるまで待機しません。

        デフォルトは NOTIFY_COMPONENTS です。
      - timeoutInSeconds 各コンポーネントが DeferComponentUpdate IPC 操作による更新通知に応答する必要がある時間枠 (秒単位)。コンポーネントがこの時間内に応答しなかった場合、コアデバイスでデプロイが実行されます。

        デフォルトは 60 秒です。

   2. (オプション) 設定検証ポリシー (configurationValidationPolicy) を指定します。このポリシーは、各コンポーネントがデプロイからの設定更新を検証する必要がある期間を定義します。コンポーネントは SubscribeToValidateConfigurationUpdates IPC 操作を使用して、自身に対する設定更新の通知にサブスクライブすることができます。これにより、コンポーネントが SendConfigurationValidityReport IPC 操作を使用して、設定更新が有効なものかどうかを AWS IoT Greengrass Core ソフトウェアに伝えられるようになります。設定更新が有効でない場合、デプロイは失敗します。

のポリシーは、次のパラメータを使用するオブジェクトです。

- timeoutInSeconds (オプション) 各コンポーネントが設定更新を検証する必要がある時間 (秒単位)。コンポーネントがこの時間内に応答しなかった場合、コアデバイスでデプロイが実行されます。

  デフォルトは 30 秒です。

3. (オプション) 障害処理ポリシー (failureHandlingPolicy) を指定します。このポリシーは、デプロイが失敗した場合にデバイスをロールバックするかどうかを定義する文字列です。次のオプションから選択します。
   - ROLLBACK - コアデバイスでデプロイが失敗した場合、AWS IoT Greengrass Core ソフトウェアは、そのコアデバイスを以前の設定にロールバックします。
   - DO_NOTHING - コアデバイスでデプロイが失敗した場合、AWS IoT Greengrass Core ソフトウェアは、新しい設定を維持します。この場合、新しい設定が有効でない場合には、コンポーネントが壊れる可能性があります。

   デフォルトは ROLLBACK です。

deployment.json でのデプロイは次の例のようになります。

```
{
  "targetArn": "targetArn",
  "deploymentName": "deploymentName",
  "components": {
    "com.example.IndustrialDashboard": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "reset": [
          "/network/useHttps",
          "/tags"
        ],
        "merge": "{\"tags\":[\"/boiler/1/temperature\",\"/boiler/1/pressure\",\"/
boiler/2/temperature\",\"/boiler/2/pressure\"]}"
      }
    }
  },
  "deploymentPolicies": {
    "componentUpdatePolicy": {
      "action": "NOTIFY_COMPONENTS",
      "timeoutInSeconds": 30
```

```
    },
    "configurationValidationPolicy": {
      "timeoutInSeconds": 60
    },
    "failureHandlingPolicy": "ROLLBACK"
  }
}
```

6. (オプション) デプロイの停止、ロールアウト、タイムアウトの方法を定義します。AWS IoT Greengrass は AWS IoT Core ジョブを使用してコアデバイスにデプロイを送信するため、これらのオプションは AWS IoT Core ジョブ用の設定オプションと同じになります。詳細については、「AWS IoT デベロッパーガイド」の「ジョブのロールアウトと中止設定」を参照してください。

   ジョブオプションを定義するには、`iotJobConfiguration` オブジェクトを `deployment.json` に追加します。次に、設定するオプションを定義します。

   `deployment.json` でのデプロイは次の例のようになります。

```
{
  "targetArn": "targetArn",
  "deploymentName": "deploymentName",
  "components": {
    "com.example.IndustrialDashboard": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "reset": [
          "/network/useHttps",
          "/tags"
        ],
        "merge": "{\"tags\":[\"/boiler/1/temperature\",\"/boiler/1/pressure\",\"/
boiler/2/temperature\",\"/boiler/2/pressure\"]}"
      }
    }
  },
  "deploymentPolicies": {
    "componentUpdatePolicy": {
      "action": "NOTIFY_COMPONENTS",
      "timeoutInSeconds": 30
    },
    "configurationValidationPolicy": {
      "timeoutInSeconds": 60
    },
```

```
    "failureHandlingPolicy": "ROLLBACK"
  },
  "iotJobConfiguration": {
    "abortConfig": {
      "criteriaList": [
        {
          "action": "CANCEL",
          "failureType": "ALL",
          "minNumberOfExecutedThings": 100,
          "thresholdPercentage": 5
        }
      ]
    },
    "jobExecutionsRolloutConfig": {
      "exponentialRate": {
        "baseRatePerMinute": 5,
        "incrementFactor": 2,
        "rateIncreaseCriteria": {
          "numberOfNotifiedThings": 10,
          "numberOfSucceededThings": 5
        }
      },
      "maximumPerMinute": 50
    },
    "timeoutConfig":  {
      "inProgressTimeoutInMinutes": 5
    }
  }
}
```

7. (オプション) タグ (tags) をデプロイに追加します。詳細については、「AWS IoT Greengrass Version 2 リソースのタグ付け」を参照してください。
8. 以下のコマンドを実行して、deployment.json からデプロイを作成します。

```
aws greengrassv2 create-deployment --cli-input-json file://deployment.json
```

レスポンスには、このデプロイを識別する deploymentId が含まれます。デプロイ ID を使用して、デプロイのステータスを確認できます。詳細については、「デプロイのステータスを確認する」を参照してください。

コンポーネント設定の更新

コンポーネント設定は、各コンポーネントのパラメータを定義する JSON オブジェクトです。各コンポーネントの recipe は、コアデバイスにコンポーネントをデプロイするときに変更するデフォルト設定を定義します。

デプロイを作成するとき、各コンポーネントに適用する設定の更新を指定できます。設定の更新はパッチ操作であり、更新がコアデバイスに存在するコンポーネント設定を修正することを意味します。コアデバイスにコンポーネントがない場合、設定更新がそのデプロイのデフォルト設定を修正して適用します。

設定更新は、リセット更新とマージ更新を定義します。リセット更新は、デフォルトにリセットまたは削除する設定値を定義します。マージ更新は、コンポーネントに設定する新しい設定値を定義します。設定更新をデプロイすると、 AWS IoT Greengrass Core ソフトウェアはマージ更新の前にリセット更新を実行します。

コンポーネントは、デプロイする設定更新を検証できます。コンポーネントは、デプロイが設定を変更した際に通知を受信するようにサブスクライブして、サポートしていない設定を拒否できます。詳細については、「コンポーネント設定とやり取り」を参照してください。

トピック

更新のリセット

リセット更新は、コアデバイスでデフォルトにリセットする設定値を定義します。設定値にデフォルト値がない場合、リセット更新がコンポーネントの設定からその値を削除します。これにより、無効な設定によって破損するコンポーネントを修正するうえで役立ちます。

JSON ポインタのリストを使用して、リセットする設定値を定義します。JSON ポインタはフォワードスラッシュ (/) で始まります。ネストされたコンポーネント設定の値を識別するには、フォワードスラッシュ (/) を使用して、設定の各レベルのキーを区切ります。詳細については、「JSON ポインタの仕様」を参照してください。

> **Note**
>
> リスト全体のみをデフォルト値にリセットできます。更新リセットを使用して、リストの個々の要素をリセットすることはできません。

コンポーネントの設定を全体的にデフォルト値にリセットするには、リセット更新として空の文字列を 1 つ指定します。

```
"reset": [""]
```

マージの更新

マージ更新は、コアのコンポーネント設定に挿入する設定値を定義します。マージ更新は、 AWS IoT Greengrass Core ソフトウェアがリセット更新で指定したパスの値をリセットした後にマージする JSON オブジェクトです。 AWS CLI または AWS SDKsを使用する場合は、この JSON オブジェクトを文字列としてシリアル化する必要があります。

コンポーネントのデフォルト設定に存在しないキー値のペアをマージできます。同じキーの値とは異なるタイプのキー値のペアをマージすることもできます。古い値は新しい値により上書きされます。つまり、設定オブジェクトの構造を変更できます。

Null 値を空の文字列、リスト、オブジェクトとマージできます。

> **Note**
>
> マージ更新は、リストに要素の挿入または追加する目的として使用することはできません。リスト全体を置き換え、あるいは各要素に一意のキーを持つオブジェクトを定義できます。AWS IoT Greengrass は設定値に JSON を使用します。JSON は数値タイプを指定しますが、整数と浮動小数点数を区別しません。その結果、 AWS IoT Greengrassで設定値が浮動小数点数に変換されることがあります。コンポーネントが正しいデータタイプを使用することを確認するには、数値の設定値を文字列として定義することをお勧めします。次に、整数または浮動小数点としてコンポーネントでパースします。これにより、設定値が設定とコアデバイスに対して同じタイプであることを保証します。

マージ更新で recipe 変数を使用する

この機能は、Greengrass nucleus コンポーネントの v2.6.0 以降で利用できます。

Greengrass nucleus interpolateComponentConfigurationの設定オプションを に設定した場合true、マージ更新で recipe 変数以外の *component_dependency_name*:configuration:*json_pointer* recipe 変数を使用できます。例えば、マージ更新で {iot:thingName} recipe 変数を使用して、コアデバイスの AWS IoT モノの名前をプロセス間通信 (IPC) 承認ポリシーなどのコンポーネント設定値に含めることができます。

例

次の例では、次のデフォルト設定を持つダッシュボードコンポーネントの設定更新を示しています。このコンポーネントの例は、産業機器に関する情報を表示します。

```
{
  "name": null,
  "mode": "REQUEST",
  "network": {
    "useHttps": true,
    "port": {
      "http": 80,
      "https": 443
    },
  },
  "tags": []
}
```

産業用ダッシュボード コンポーネント recipe

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.IndustrialDashboard",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Displays information about industrial equipment.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "name": null,
      "mode": "REQUEST",
      "network": {
        "useHttps": true,
        "port": {
          "http": 80,
```

```
          "https": 443
        },
      },
      "tags": []
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "run": "python3 -u {artifacts:path}/industrial_dashboard.py"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "run": "py -3 -u {artifacts:path}/industrial_dashboard.py"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.IndustrialDashboard
ComponentVersion: '1.0.0'
ComponentDescription: Displays information about industrial equipment.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    name: null
    mode: REQUEST
    network:
      useHttps: true
      port:
        http: 80
        https: 443
```

```
    tags: []
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      run: |
        python3 -u {artifacts:path}/industrial_dashboard.py
  - Platform:
      os: windows
    Lifecycle:
      run: |
        py -3 -u {artifacts:path}/industrial_dashboard.py
```

Example 例 1: マージ更新

次の設定更新を適用するデプロイを作成します。この更新は、マージ更新を指定しますが、リセット更新は指定しません。この設定更新は、2 基のボイラーのデータで HTTP ポート 8080 にダッシュボードを表示するように、コンポーネントに指示します。

Console

マージする設定

```
{
  "name": "Factory 2A",
  "network": {
    "useHttps": false,
    "port": {
      "http": 8080
    }
  },
  "tags": [
    "/boiler/1/temperature",
    "/boiler/1/pressure",
    "/boiler/2/temperature",
    "/boiler/2/pressure"
  ]
}
```

AWS CLI

次のコマンドは、コアデバイスにデプロイを作成します。

```
aws greengrassv2 create-deployment --cli-input-json file://dashboard-deployment.json
```

dashboard-deployment.json ファイルには、次の JSON ドキュメントが含まれています。

```
{
  "targetArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
  "deploymentName": "Deployment for MyGreengrassCore",
  "components": {
    "com.example.IndustrialDashboard": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "merge": "{\"name\":\"Factory 2A\",\"network\":{\"useHttps\":false,\"port
\":{\"http\":8080}},\"tags\":[\"/boiler/1/temperature\",\"/boiler/1/pressure\",\"/
boiler/2/temperature\",\"/boiler/2/pressure\"]}"
      }
    }
  }
}
```

Greengrass CLI

次の Greengrass CLI コマンドは、コアデバイスにローカルデプロイを作成します。

```
sudo greengrass-cli deployment create \
  --recipeDir recipes \
  --artifactDir artifacts \
  --merge "com.example.IndustrialDashboard=1.0.0" \
  --update-config dashboard-configuration.json
```

dashboard-configuration.json ファイルには、次の JSON ドキュメントが含まれています。

```
{
  "com.example.IndustrialDashboard": {
    "MERGE": {
      "name": "Factory 2A",
      "network": {
```

```
        "useHttps": false,
        "port": {
          "http": 8080
        }
      },
      "tags": [
        "/boiler/1/temperature",
        "/boiler/1/pressure",
        "/boiler/2/temperature",
        "/boiler/2/pressure"
      ]
    }
  }
}
```

この更新の後、ダッシュボードコンポーネントは次の設定になります:

```
{
  "name": "Factory 2A",
  "mode": "REQUEST",
  "network": {
    "useHttps": false,
    "port": {
      "http": 8080,
      "https": 443
    }
  },
  "tags": [
    "/boiler/1/temperature",
    "/boiler/1/pressure",
    "/boiler/2/temperature",
    "/boiler/2/pressure"
  ]
}
```

Example 例 2: 更新のリセットとマージ

次に、次の設定更新を適用するデプロイを作成します。この更新は、リセット更新とマージ更新を指定します。これらの更新は、デフォルトの HTTPS ポートに異なるボイラーのデータでダッシュボードを表示するように指定します。これらの更新は、前の例で示されている設定更新の結果となる設定を変更します。

Console

パスのリセット

```
[
  "/network/useHttps",
  "/tags"
]
```

マージする設定

```
{
  "tags": [
    "/boiler/3/temperature",
    "/boiler/3/pressure",
    "/boiler/4/temperature",
    "/boiler/4/pressure"
  ]
}
```

AWS CLI

次のコマンドは、コアデバイスにデプロイを作成します。

```
aws greengrassv2 create-deployment --cli-input-json file://dashboard-
deployment2.json
```

dashboard-deployment2.json ファイルには、次の JSON ドキュメントが含まれています。

```
{
  "targetArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
  "deploymentName": "Deployment for MyGreengrassCore",
  "components": {
    "com.example.IndustrialDashboard": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "reset": [
          "/network/useHttps",
          "/tags"
        ],
```

```
        "merge": "{\"tags\":[\"/boiler/3/temperature\",\"/boiler/3/pressure\",\"/
boiler/4/temperature\",\"/boiler/4/pressure\"]}"
      }
    }
  }
}
```

Greengrass CLI

次の Greengrass CLI コマンドは、コアデバイスにローカルデプロイを作成します。

```
sudo greengrass-cli deployment create \
  --recipeDir recipes \
  --artifactDir artifacts \
  --merge "com.example.IndustrialDashboard=1.0.0" \
  --update-config dashboard-configuration2.json
```

dashboard-configuration2.json ファイルには、次の JSON ドキュメントが含まれています。

```
{
  "com.example.IndustrialDashboard": {
    "RESET": [
      "/network/useHttps",
      "/tags"
    ],
    "MERGE": {
      "tags": [
        "/boiler/3/temperature",
        "/boiler/3/pressure",
        "/boiler/4/temperature",
        "/boiler/4/pressure"
      ]
    }
  }
}
```

この更新の後、ダッシュボードコンポーネントは次の設定になります:

```
{
  "name": "Factory 2A",
```

```
  "mode": "REQUEST",
  "network": {
    "useHttps": true,
    "port": {
      "http": 8080,
      "https": 443
    }
  },
  "tags": [
    "/boiler/3/temperature",
    "/boiler/3/pressure",
    "/boiler/4/temperature",
    "/boiler/4/pressure",
  ]
}
```

サブデプロイを作成する

> **Note**
>
> サブデプロイ機能は、Greengrass nucleus バージョン 2.9.0 以降で使用できます。Greengrass nucleus の旧コンポーネントバージョンでは、設定をサブデプロイにデプロイすることはできません。

サブデプロイは、親デプロイ内のデバイスのより小さなサブセットをターゲットとするデプロイです。サブデプロイを使用して、デバイスのより小さなサブセットに設定をデプロイできます。サブデプロイを作成して、親デプロイ内の 1 つ以上のデバイスで失敗した場合に、失敗した親デプロイを再試行することもできます。この機能を使用すると、その親デプロイで失敗したデバイスを選択し、サブデプロイを作成して、そのサブデプロイが成功するまで設定をテストできます。サブデプロイが成功したら、その設定を親デプロイに再デプロイできます。

サブデプロイを作成し、そのステータスを確認するには、このセクションのステップに従います。デプロイを作成する方法の詳細については、「デプロイの作成」を参照してください。

サブデプロイを作成するには (AWS CLI)

1. 次のコマンドを実行して、モノのグループの最新のデプロイを取得します。コマンド内の ARN を、クエリするモノグループの ARN に置き換えます。`--history-filter` を **`LATEST_ONLY`** に設定すると、そのモノのグループの最新のデプロイが表示されます。

```
aws greengrassv2 list-deployments --target-arn arn:aws:iot:region:account-
id:thinggroup/thingGroupName --history-filter LATEST_ONLY
```

2. 次のステップで使用する list-deployments コマンドに対するレスポンスから deploymentId をコピーします。
3. 次のコマンドを実行し、デプロイのステータスを取得します。*deploymentId* をクエリするデプロイの ID に置き換えます。

```
aws greengrassv2 get-deployment --deployment-id deploymentId
```

4. 次のステップで使用する get-deployment コマンドに対するレスポンスから iotJobId をコピーします。
5. 次のコマンドを実行して、指定されたジョブの実行リストを取得します。*jobID* を前のステップの iotJobId に置き換えます。*status* を、フィルタリングするステータスに置き換えます。次のステータスで結果をフィルタリングできます。

   - QUEUED
   - IN_PROGRESS
   - SUCCEEDED
   - FAILED
   - TIMED_OUT
   - REJECTED
   - REMOVED
   - CANCELED

```
aws iot list-job-executions-for-job --job-id jobID --status status
```

6. サブデプロイのために、新しい AWS IoT モノのグループを作成するか、または既存のモノのグループを使用します。その後、このモノのグループに AWS IoT モノを追加します。モノグループを使用して Greengrass コアデバイスのフリートを管理します。ソフトウェアコンポーネントをデバイスにデプロイするとき、個々のデバイスまたはデバイスのグループのいずれかを対象にできます。アクティブな Greengrass デプロイを備えたモノのグループにデバイスを追加できます。追加すると、そのモノのグループのソフトウェアコンポーネントをそのデバイスにデプロイできるようになります。

新しいモノのグループを作成して、それにデバイスを追加するには、次を実行します。

a. AWS IoT モノのグループを作成します。を新しいモノのグループの名前*MyGreengrassCoreGroup*に置き換えます。モノのグループ名にコロン (:) は使用できません。

> **Note**
>
> サブデプロイのモノのグループが 1 つの parentTargetArn で使用されている場合、それを別の親フリートで再利用することはできません。あるモノのグループが別のフリートのサブデプロイを作成するために既に使用されている場合、API は エラーを返します。

```
aws iot create-thing-group --thing-group-name MyGreengrassCoreGroup
```

リクエストが正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "thingGroupName": "MyGreengrassCoreGroup",
  "thingGroupArn": "arn:aws:iot:us-
west-2:123456789012:thinggroup/MyGreengrassCoreGroup",
  "thingGroupId": "4df721e1-ff9f-4f97-92dd-02db4e3f03aa"
}
```

b. プロビジョンド Greengrass コアをモノのグループに追加します。次のパラメータを使用して次のコマンドを実行します。

- をプロビジョニングされた Greengrass コアの名前*MyGreengrassCore*に置き換えます。
- をモノのグループの名前*MyGreengrassCoreGroup*に置き換えます。

```
aws iot add-thing-to-thing-group --thing-name MyGreengrassCore --thing-group-
name MyGreengrassCoreGroup
```

要求が正常に処理された場合、コマンドは出力されません。

7. deployment.json という名前のファイルを作成して、次の JSON オブジェクトをファイルにコピーします。*targetArn* をサブデプロイがターゲットとする AWS IoT モノのグループの ARN に置き換えます。サブデプロイのターゲットにできるのは、モノのグループのみです。モノのグループの ARN の形式は、次のとおりです。

   - モノのグループ – arn:aws:iot:*region*:*account-id*:thinggroup/*thingGroupName*

   ```
   {
     "targetArn": "targetArn"
   }
   ```

8. 次のコマンドを再度実行して、元のデプロイの詳細を取得します。これらの詳細には、メタデータ、コンポーネント、ジョブ構成が含まれます。*deploymentId* を Step 1 の ID に置き換えます。このデプロイ設定を使用してサブデプロイを設定し、必要に応じて変更できます。

   ```
   aws greengrassv2 get-deployment --deployment-id deploymentId
   ```

   レスポンスには、デプロイの詳細が含まれています。get-deployment コマンドのレスポンスにある次のキーと値のペアを deployment.json にコピーします。これらの値を、サブデプロイ向けに変更することができます。このコマンドの詳細については、「」を参照してくださいGetDeployment。

   - components - デプロイのコンポーネント。コンポーネントをアンインストールする場合は、このオブジェクトから削除してください。
   - deploymentName - デプロイの名前。
   - deploymentPolicies - デプロイのポリシー。
   - iotJobConfiguration - デプロイのジョブ設定。
   - parentTargetArn – 親デプロイのターゲット。
   - tags - デプロイのタグ。

9. 次のコマンドを実行して、deployment.json からサブデプロイを作成します。*subdeploymentName* をサブデプロイの名前に置き換えます。

   ```
   aws greengrassv2 create-deployment --deployment-name subdeploymentName --cli-input-
   json file://deployment.json
   ```

レスポンスには、このサブデプロイを識別する deploymentId が含まれます。デプロイ ID を使用して、デプロイのステータスを確認できます。詳細については、「デプロイのステータスを確認する」を参照してください。

10. サブデプロイが成功した場合は、その設定を使用して親デプロイを修正できます。前のステップで使用した deployment.json をコピーします。JSON ファイルの targetArn を親デプロイの ARN に置き換え、次のコマンドを実行してこの新しい設定で親デプロイを作成します。

> **Note**
>
> 親フリートの新しいデプロイリビジョンを作成すると、その親デプロイのすべてのデプロイリビジョンとサブデプロイが置き換えられます。詳細については、「デプロイの変更」を参照してください。

```
aws greengrassv2 create-deployment --cli-input-json file://deployment.json
```

レスポンスには、このデプロイを識別する deploymentId が含まれます。デプロイ ID を使用して、デプロイのステータスを確認できます。詳細については、「デプロイのステータスを確認する」を参照してください。

展開の改訂

各ターゲットのモノまたはモノグループは、一度に 1 つのアクティブなデプロイを持つことができます。既にデプロイがあるターゲットのデプロイを作成するとき、新しいデプロイのソフトウェアコンポーネントが、以前のデプロイのソフトウェアコンポーネントを置き換えます。新しいデプロイが、以前のデプロイが定義するコンポーネントを定義しない場合、AWS IoT Greengrass Core ソフトウェアはターゲットコアデバイスからそのコンポーネントを削除します。既存のデプロイを改訂して、ターゲットに対する以前のデプロイのコアデバイスで実行されるコンポーネントを削除しないようにできます。

デプロイを改訂するには、以前のデプロイに存在する同じコンポーネントと設定から開始するデプロイを作成します。CreateDeployment オペレーションを使用します。これは、デプロイの作成に使用するのと同じオペレーションです。

デプロイ (AWS CLI) を改訂するには

1. 次のコマンドを実行して、デプロイターゲットのデプロイを一覧表示します。*targetArn* をターゲット AWS IoT モノまたはモノグループの ARN に置き換えます。

   ```
   aws greengrassv2 list-deployments --target-arn targetArn
   ```

   レスポンスには、ターゲットの最新デプロイのリストが含まれています。deploymentId を次のステップで使用するレスポンスからコピーします。

   > **Note**
   >
   > ターゲットの最新リビジョン以外のデプロイを修正することもできます。--history-filter ALL 引数を指定して、ターゲットのすべてのデプロイを一覧表示します。次に、修正するデプロイの ID をコピーします。

2. 次のコマンドを実行して、デプロイの詳細を取得します。これらの詳細には、メタデータ、コンポーネント、ジョブ設定が含まれます。*deploymentId* を前のステップの ID に置き換えます。

   ```
   aws greengrassv2 get-deployment --deployment-id deploymentId
   ```

   レスポンスには、デプロイの詳細が含まれています。

3. deployment.json という名前のファイルを作成し、前のコマンドのレスポンスをファイルにコピーします。
4. deployment.json の JSON オブジェクトから次のキーと値のペアを削除します。

   - deploymentId
   - revisionId
   - iotJobId
   - iotJobArn
   - creationTimestamp
   - isLatestForTarget
   - deploymentStatus

   CreateDeployment オペレーションでは、次の構造のペイロードを想定しています。

```
{
  "targetArn": "String",
  "components": Map of components,
  "deploymentPolicies": DeploymentPolicies,
  "iotJobConfiguration": DeploymentIoTJobConfiguration,
  "tags": Map of tags
}
```

5. deployment.json で、次のいずれかを行ってください。
   - デプロイの名前 (deploymentName) を変更します。
   - デプロイのコンポーネント (components) を変更します。
   - デプロイのポリシー (deploymentPolicies) を変更します。
   - デプロイのジョブ設定 (iotJobConfiguration) を変更します。
   - デプロイのタグ (tags) を変更します。

   これらのデプロイを定義する方法の詳細については、「デプロイの作成」を参照してください。
6. 以下のコマンドを実行して、deployment.json からデプロイを作成します。

```
aws greengrassv2 create-deployment --cli-input-json file://deployment.json
```

   レスポンスには、このデプロイを識別する deploymentId が含まれます。デプロイ ID を使用して、デプロイのステータスを確認できます。詳細については、「デプロイのステータスを確認する」を参照してください。

デプロイをキャンセルする

アクティブなデプロイをキャンセルして、AWS IoT Greengrass コアデバイスにソフトウェアコンポーネントがインストールされるのを防ぐことができます。モノグループをターゲットとするデプロイをキャンセルすると、グループに追加したコアデバイスが、その継続的デプロイを受信しなくなります。コアデバイスで既にデプロイが実行されている場合、デプロイをキャンセルしても、そのデバイスのコンポーネントは変更されません。キャンセルされたデプロイを受信したコアデバイス上で実行されるコンポーネントを修正するには、新しいデプロイを作成するか、デプロイを修正する必要があります。

デプロイをキャンセルするには (AWS CLI)

1. 次のコマンドを実行し、ターゲットの最新のデプロイリビジョンの ID を検索します。新しいリビジョンの作成時に以前のデプロイはキャンセルされるため、最新のリビジョンが、ターゲットに対してアクティブにできる唯一のデプロイとなります。*targetArn* をターゲット AWS IoT モノまたはモノグループの ARN に置き換えます。

   ```
   aws greengrassv2 list-deployments --target-arn targetArn
   ```

   レスポンスには、ターゲットの最新デプロイのリストが含まれています。deploymentId を次のステップで使用するレスポンスからコピーします。

2. 以下のコマンドを実行して、デプロイをキャンセルします。*deploymentId* を前のステップの ID に置き換えます。

   ```
   aws greengrassv2 cancel-deployment --deployment-id deploymentId
   ```

   操作が成功すると、デプロイのステータスが CANCELED に変わります。

デプロイのステータスを確認する

AWS IoT Greengrass で作成したデプロイのステータスを確認できます。また、各コアデバイスにデプロイをロールアウトする AWS IoT ジョブのステータスも確認できます。デプロイがアクティブな間は、AWS IoT ジョブのステータスは IN_PROGRESS になります。デプロイの新しいリビジョンを作成すると、前のリビジョンの AWS IoT ジョブのステータスが CANCELLED に変わります。

トピック

- デプロイのステータスを確認する
- デバイスへのデプロイのステータスを確認する

デプロイのステータスを確認する

ターゲットまたはその ID で識別したデプロイのステータスを確認できます。

ターゲットでデプロイステータスを確認するには (AWS CLI)

- 次のコマンドを実行し、ターゲットの最新のデプロイの ID を検索します。*targetArn* は、デプロイがターゲットとする AWS IoT のモノまたはモノグループの Amazon リソースネーム (ARN) に置き換えます。

  ```
  aws greengrassv2 list-deployments --target-arn targetArn
  ```

  レスポンスには、ターゲットの最新デプロイのリストが含まれています。このデプロイオブジェクトには、デプロイのステータスが含まれます。

ID でデプロイステータスを確認するには (AWS CLI)

- 次のコマンドを実行し、デプロイのステータスを取得します。*deploymentId* をクエリするデプロイの ID に置き換えます。

  ```
  aws greengrassv2 get-deployment --deployment-id deploymentId
  ```

  レスポンスには、デプロイのステータスが含まれます。

デバイスへのデプロイのステータスを確認する

個々のコアデバイスに適用されるデプロイジョブのステータスを確認できます。また、モノのグループのデプロイで、デプロイジョブのステータスを確認することもできます。

コアデバイスへのデプロイジョブのステータスを確認するには (AWS CLI)

- 次のコマンドを実行し、コアデバイスに対するすべてのデプロイジョブのステータスを取得します。*coreDeviceName* をクエリするコアデバイスの名前に置き換えます。

  ```
  aws greengrassv2 list-effective-deployments --core-device-thing-name coreDeviceName
  ```

  レスポンスには、コアデバイスのデプロイジョブのリストが含まれます。デプロイのジョブは、ジョブの `deploymentId` または `targetArn` により識別できます。各デプロイジョブには、コアデバイス上のジョブのステータスが含まれます。

モノのグループのデプロイステータスを確認するには (AWS CLI)

1. 次のコマンドを実行し、既存のデプロイの ID を取得します。*targetArn* は、ターゲットであるモノのグループの ARN に置き換えます。

   ```
   aws greengrassv2 list-deployments --target-arn targetArn
   ```

   レスポンスには、ターゲットの最新デプロイのリストが含まれています。deploymentId を次のステップで使用するレスポンスからコピーします。

   > **Note**
   >
   > ターゲットの最新デプロイ以外のデプロイを一覧表示することもできます。--history-filter ALL 引数を指定して、ターゲットのすべてのデプロイを一覧表示します。次に、ステータスを確認するデプロイの ID をコピーします。

2. 次のコマンドを実行して、デプロイの詳細を取得します。*deploymentID* を前のステップで取得した ID に置き換えます。

   ```
   aws greengrassv2 get-deployment --deployment-id deploymentId
   ```

   このレスポンスには、デプロイに関する情報が含まれています。次のステップで使用するために、レスポンスから iotJobId をコピーします。

3. 次のコマンドを実行し、デプロイにおけるコアデバイスのジョブ実行について詳細を表示します。*iotJobId* と *coreDeviceThingName* は、前のステップで取得したジョブ ID 、およびステータスを確認するコアデバイスに置き換えます。

   ```
   aws iot describe-job-execution --job-id iotJobId --thing-name coreDeviceThingName
   ```

   このレスポンスには、コアデバイスにおけるデプロイジョブの実行ステータスと、ステータスに関する詳細情報が含まれます。detailsMap には、以下の情報が含まれています。

   - detailed-deployment-status – デプロイ結果のステータスで、以下のいずれかの値を取ります。
     - SUCCESSFUL – デプロイは成功しました。
     - FAILED_NO_STATE_CHANGE – コアデバイスがデプロイの適用を準備をしている間に、そのデプロイが失敗しました。

- FAILED_ROLLBACK_NOT_REQUESTED – デプロイは失敗しました。このデプロイでは、以前の動作構成にロールバックするための指定が行われていないため、コアデバイスが正しく機能していない可能性があります。
- FAILED_ROLLBACK_COMPLETE – デプロイは失敗しましたが、コアデバイスは以前の動作設定に正常にロールバックされました。
- FAILED_UNABLE_TO_ROLLBACK – デプロイが失敗しました。コアデバイスは、以前の動作設定にロールバックできなかったため、正しく機能していない可能性があります。

デプロイが失敗した場合は、deployment-failure-cause 値、およびコアデバイスのログファイルにより問題を特定します。コアデバイスのログファイルへのアクセス方法については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

- deployment-failure-cause – ジョブの実行が失敗した理由について、追加的な情報を提供するエラーメッセージです。

このレスポンスは、次の例のようになります。

```
{
  "execution": {
    "jobId": "2cc2698a-5175-48bb-adf2-1dd345606ebd",
    "status": "FAILED",
    "statusDetails": {
      "detailsMap": {
        "deployment-failure-cause": "No local or cloud component version
 satisfies the requirements. Check whether the version constraints conflict and
 that the component exists in your AWS ##### with a version that matches the
 version constraints. If the version constraints conflict, revise deployments
 to resolve the conflict. Component com.example.HelloWorld version constraints:
 LOCAL_DEPLOYMENT requires =1.0.0, thinggroup/MyGreengrassCoreGroup requires
 =1.0.1.",
        "detailed-deployment-status": "FAILED_NO_STATE_CHANGE"
      }
    },
    "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
    "queuedAt": "2022-02-15T14:45:53.098000-08:00",
    "startedAt": "2022-02-15T14:46:05.670000-08:00",
    "lastUpdatedAt": "2022-02-15T14:46:20.892000-08:00",
    "executionNumber": 1,
    "versionNumber": 3
  }
```

```
}
```

AWS IoT Greengrass でのログ記録とモニタリング

モニタリングは、AWS IoT Greengrass と AWS ソリューションの信頼性、可用性、パフォーマンスを維持する上で重要な部分です。マルチポイント障害が発生した場合は、その障害をより簡単にデバッグできるように、AWS ソリューションのすべての部分からモニタリングデータを収集する必要があります。ただし、AWS IoT Greengrass のモニタリングをスタートする前に、以下の質問に対する回答を反映したモニタリング計画を作成する必要があります。

- どのような目的でモニタリングしますか?
- どのリソースをモニタリングしますか?
- どのくらいの頻度でこれらのリソースをモニタリングしますか?
- どのモニタリングツールを使用しますか?
- 誰がモニタリングタスクを実行しますか?
- 問題が発生したときに誰が通知を受け取りますか?

トピック

モニタリングツール

AWS では、 のモニタリングに使用できるツールを提供しています。AWS IoT Greengrass自動的にモニタリングが行われるように、これらのツールを設定できます。手動操作を必要とするツールもあります。モニタリングタスクをできるだけ自動化することをお勧めします。

次の自動化されたモニタリングツールを使用して、AWS IoT Greengrass をモニタリングして問題を報告できます。

- Amazon CloudWatch Logs – AWS CloudTrailまたはその他のソースからのログファイルをモニタリング、保存、およびアクセスします。詳細については、「Amazon ユーザーガイド」の「ログファイルのモニタリング」を参照してください。 CloudWatch
- AWS CloudTrail ログのモニタリング – アカウント間でログファイルを共有し、 CloudWatch ログにログを送信してリアルタイムで CloudTrail ログファイルをモニタリングし、Java でログ処理アプリケーションを記述して、 による配信後にログファイルが変更されていないことを確認します CloudTrail。詳細については、「 ユーザーガイド」の CloudTrail 「ログファイルの使用AWS CloudTrail」を参照してください。
- Greengrass システム ヘルス テレメトリー - Greengrass コアから送信されたテレメトリデータを受信するためにサブスクライブします。詳細については、「the section called “システムヘルステレメトリデータを収集する”」を参照してください。
- デバイスヘルス通知 Amazon を使用してイベントを作成し EventBridge 、デプロイとコンポーネントに関するステータス更新を受信します。詳細については、「デプロイとコンポーネントのヘルスステータス通知を受け取る」を参照してください。
- フリートステータスサービス - フリートステータス API 操作を使用して、コアデバイスとその Greengrass コンポーネントのステータスを確認します。AWS IoT Greengrass コンソールでフリートステータス情報を確認することもできます。詳細については、「Greengrass コアデバイスのステータスを確認する」を参照してください。

AWS IoT Greengrass ログのモニタリング

AWS IoT Greengrass は、クラウドサービスと AWS IoT Greengrass Core ソフトウェアで設定されます。AWS IoT Greengrass Core ソフトウェアは、Amazon CloudWatch Logs とコアデバイスのローカルファイルシステムにログを書き込むことができます。コアデバイスで実行される Greengrass コンポーネントは、 CloudWatch Logs とローカルファイルシステムにログを書き込むこともできます。問題をトラブルシューティングするには、ログを使用してイベントをモニタリングします。AWS IoT Greengrass ログエントリにはすべて、タイムスタンプ、ログレベル、イベントに関する情報が含まれています。

デフォルトで、AWS IoT Greengrass Core ソフトウェアはローカルファイルシステムのみにログを書き込みます。リアルタイムでファイル システム ログを確認できるため、開発とデプロイする Greengrass コンポーネントをデバッグできます。 CloudWatch Logs にログを書き込むようにコアデバイスを設定して、ローカルファイルシステムにアクセスせずにコアデバイスをトラブルシューティングすることもできます。詳細については、「 CloudWatch ログへのログ記録を有効にする」を参照してください。

トピック

ファイル システム ログをアクセス

AWS IoT Greengrass Core ソフトウェアは、コアデバイスの */greengrass/v2*/logsフォルダにログを保存します。ここで、*/greengrass/v2*はAWS IoT Greengrassルートフォルダへのパスです。ログフォルダは次の構造があります。

```
/greengrass/v2
### logs
    ### greengrass.log
    ### greengrass_2021_09_14_15_0.log
    ### ComponentName.log
    ### ComponentName_2021_09_14_15_0.log
    ### main.log
```

- `greengrass.log` - AWS IoT Greengrass Core ソフトウェアのログファイル。このログファイルを使用して、コンポーネントとデプロイに関するリアルタイム情報を確認します。このログファイルには、Greengrass nucleus のログが含まれています。Greengrass nucleus は AWS IoT Greengrass Core ソフトウェアとプラグインコンポーネント (ログマネージャーとシークレットマネージャーなど) のコアです。
- *ComponentName*`.log` - Greengrass コンポーネントのログファイル。コンポーネント ログ ファイルを使用して、コアデバイスに実行される Greengrass コンポーネントに関するリアルタイム情報を確認します。ジェネリックコンポーネントと Lambda コンポーネントは、標準出力 (stdout) と標準エラー (stderr) をこれらのログファイルに書き込みます。
- `main.log` - コンポーネントライフサイクルを処理する `main` サービスのログファイル。このログファイルは常に空の状態になります。

プラグイン、ジェネリック、Lambda コンポーネントの違いの詳細については、「コンポーネントタイプ」を参照してください。

以下の考慮事項は、ファイルシステムログを使用する場合に適用されます。

- ルートユーザーの許可

  ファイルシステムの AWS IoT Greengrass ログを読み取る root 権限が必要です。

- ログファイルのローテーション

  AWS IoT Greengrass Core ソフトウェアは、1 時間ごと、またはファイルサイズの制限を超えたときにログファイルをローテーションします。ローテーションされたログファイルは、ファイル名にタイムスタンプが含まれています。例えば、ローテーションした AWS IoT Greengrass Core ソフトウェアのログファイルは `greengrass_2021_09_14_15_0.log` という名前が付けられる場合があります。デフォルトのファイルサイズの制限は 1,024 KB (1 MB) です。ファイルサイズの制限は、Greengrass nucleus コンポーネントで設定できます。

- ログファイルの削除

  AWS IoT Greengrass Core ソフトウェアは、AWS IoT Greengrass Core ソフトウェアのログファイルまたは Greengrass コンポーネントのログファイルのサイズ (ローテーションされたログファイルを含む) がディスク容量の上限を超えたとき、以前のログファイルをクリーンアップします。AWS IoT Greengrass Core ソフトウェア ログと各コンポーネントログ のデフォルトのディスク容量上限は、10,240 KB (10 MB) です。Greengrass nucleus コンポーネントまたはログマネージャーコンポーネントの AWS IoT Greengrass Core ソフトウェアログのディスク容量上限を設定できます。ログマネージャーコンポーネントで、各コンポーネントのログディスク容量上限を設定できます。

AWS IoT Greengrass Core ソフトウェアのログファイルを確認するには

- 次のコマンドを実行して、ログファイルをリアルタイムで確認します。をAWS IoT Greengrass ルートフォルダへのパス*/greengrass/v2*に置き換えます。

  Linux or Unix

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\logs\com.example.HelloWorld.log
```

type コマンドは、ファイルのコンテンツを端末に書き込みます。このコマンドを複数回実行して、ファイル内の変更を確認してください。

PowerShell

```
gc C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
```

コンポーネントのログファイルを確認するには

- 次のコマンドを実行して、ログファイルをリアルタイムで確認します。*/greengrass/v2* または *C:\greengrass\v2* をAWS IoT Greengrassルートフォルダへのパスに置き換え、*com.example.HelloWorld* をコンポーネントの名前に置き換えます。

  Linux or Unix

  ```
  sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
  ```

  PowerShell

  ```
  gc C:\greengrass\v2\logs\com.example.HelloWorld.log -Tail 10 -Wait
  ```

Greengrass CLI の logs コマンドを使用して、コアデバイスの Greengrass ログを分析することもできます。logs コマンドを実行するには、Greengrass nucleus が JSON 形式のログファイルを出力するように設定する必要があります。詳細については、「Greengrass コマンドラインインターフェイス と ログ」を参照してください。

アクセス CloudWatch ログ

ログマネージャーコンポーネントをデプロイして、 CloudWatch ログに書き込むようにコアデバイスを設定できます。詳細については、「 CloudWatch ログへのログ記録を有効にする」を参照してください。その後、Amazon CloudWatch コンソールのログページで、または Logs API CloudWatch を使用してログを表示できます。

ロググループ名

```
/aws/greengrass/componentType/region/componentName
```

ロググループ名は、次の変数を使用します。

- componentType - 次のいずれかに該当するコンポーネントのタイプ。
  - GreengrassSystemComponent — このロググループには、Greengrass nucleus と同じ JVM で実行される nucleus とプラグインコンポーネントのログが含まれます。コンポーネントは Greengrass nucleus の一部です。
  - UserComponent – このロググループには、ジェネリックコンポーネント、Lambda コンポーネント、およびデバイス上の他のアプリケーションのログが含まれます。コンポーネントは Greengrass nucleus の一部ではありません。

  詳細については、「コンポーネントタイプ」を参照してください。
- region - コアデバイスが使用する AWS リージョン。
- componentName - コンポーネントの名前。システムログの場合、この値は System です。

ログストリーム名

```
/date/thing/thingName
```

ログストリーム名は次の変数を使用します。

- date - 2020/12/15 など、ログの日付。ログマネージャーコンポーネントは yyyy/MM/dd 形式を使用します。
- thingName - コアデバイスの名前。

> **Note**
>
> モノの名前にコロン (:) が含まれている場合、ログマネージャーはコロンをプラス (+) に置き換えます。

ログマネージャーコンポーネントを使用して CloudWatch Logs に書き込む場合は、次の考慮事項が適用されます。

- ログ遅延

> **Note**
>
> ログマネージャーバージョン 2.3.0 にアップグレードすることをお勧めします。これにより、ローテーションされたアクティブなログファイルのログ遅延が軽減されます。ログマネージャー 2.3.0 にアップグレードする際には、Greengrass nucleus 2.9.1 にもアップグレードすることをお勧めします。

ログマネージャーコンポーネントバージョン 2.2.8 (およびそれ以前) は、ローテーションされたログファイルからのみ、ログを処理およびアップロードします。デフォルトで、AWS IoT Greengrass Core ソフトウェアは、1 時間ごと、または 1,024 KB に達した後に、ログファイルをローテーションします。その結果、ログマネージャーコンポーネントは、AWS IoT Greengrass Core ソフトウェアまたは Greengrass コンポーネントが 1,024 KB を超えるログを書き込んだ後にのみ、ログをアップロードします。ログファイルの容量上限を低く設定して、ログファイルのローテーション頻度を上げることができます。これにより、ログマネージャーコンポーネントはログをより頻繁に CloudWatch ログにアップロードします。

ログマネージャーコンポーネントバージョン 2.3.0 (およびそれ以降) は、すべてのログを処理およびアップロードします。新しいログを書き込むと、ログマネージャーバージョン 2.3.0 (およびそれ以降) は、ローテーションを待つことなく、そのアクティブなログファイルを処理して直接アップロードします。そのため、5 分以内に新しいログを表示することができます。

ログマネージャーコンポーネントは、新しいログを定期的にアップロードします。デフォルトで、ログマネージャーコンポーネントは 5 分ごとに新しいログをアップロードします。ログマネージャーコンポーネントが を設定することでログをより頻繁に CloudWatch Logs にアップロードするように、アップロード間隔を短く設定できます`periodicUploadIntervalSec`。この定期的な間隔を設定する方法の詳細については、「設定」を参照してください。

ログは、同じ Greengrass ファイルシステムからほぼリアルタイムでアップロードできます。ログをリアルタイムで監視する必要がある場合、ファイルシステムログの使用を検討してください。

> **Note**
>
> ログの書き込み先として別のファイルシステムを使用している場合、ログマネージャーは、ログマネージャーコンポーネントバージョン 2.2.8 以前の動作に戻ります。ファイル

システムログへのアクセスについては、「ファイルシステムログをアクセス」を参照してください。

- クロックスキュー

  ログマネージャーコンポーネントは、標準の Signature Version 4 署名プロセスを使用して、CloudWatch ログへの API リクエストを作成します。コアデバイスのシステム時間が同期していない時間が 15 分を超えると、 CloudWatch ログはリクエストを拒否します。詳細については、「AWS 全般のリファレンス」の「署名バージョン 4 の署名プロセス」を参照してください。

システム サービス ログにアクセス

システムサービスとして AWS IoT Greengrass Core ソフトウェアを設定する場合、システムサービスログを確認して、ソフトウェアの起動失敗など、問題をトラブルシューティングできます。

システム サービス ログ (CLI) を表示するには

1. 以下のコマンドを実行して AWS IoT Greengrass Core ソフトウェア システムのサービスログを確認します。

   Linux or Unix (systemd)

   ```
   sudo journalctl -u greengrass.service
   ```

   Windows Command Prompt (CMD)

   ```
   type C:\greengrass\v2\logs\greengrass.wrapper.log
   ```

   PowerShell

   ```
   gc C:\greengrass\v2\logs\greengrass.wrapper.log
   ```

2. Windows デバイスでは、AWS IoT Greengrass Core ソフトウェアがシステムサービスエラー用に別のログファイルを作成します。次のコマンドを実行して、システムサービスエラーログを表示します。

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\logs\greengrass.err.log
```

PowerShell

```
gc C:\greengrass\v2\logs\greengrass.err.log
```

Windows デバイスで、[Event Viewer] (イベントビューワー) アプリケーションを使用してシステムサービス ログも確認できます。

Windows サービスログ (イベントビューア) を確認するには

1. Event Viewer (イベントビューワー) アプリケーションを開きます。
2. [Windows Logs] (Windows ログ) を選択して展開します。
3. [Application] (アプリケーション) を選択して、アプリケーション サービス ログを確認します。
4. [Source] (ソース) が [greengrass] のイベントログを検索して開きます。

CloudWatch ログへのログ記録を有効にする

ログマネージャーコンポーネントをデプロイして、 CloudWatch ログにログを書き込むようにコアデバイスを設定できます。AWS IoT Greengrass Core ソフトウェア CloudWatch ログのログを有効にし、特定の Greengrass コンポーネントの CloudWatch ログを有効にできます。

> **Note**
>
> 次の IAM ポリシーの例に示すように、Greengrass コアデバイスのトークン交換ロールは、コアデバイスが CloudWatch Logs に書き込むことを許可する必要があります。自動リソースプロビジョニング機能を備えた AWS IoT Greengrass Core ソフトウェアをインストールした場合、コアデバイスにはこれらの許可が付与されています。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "logs:CreateLogGroup",
```

```
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws:logs:*:*:*"
    }
  ]
}
```

AWS IoT Greengrass Core ソフトウェアログを CloudWatch Logs に書き込むようにコアデバイスを設定するには、 `aws.greengrass.LogManager` コンポーネント`true`に対して を `uploadToCloudWatch`に設定する設定更新を指定するデプロイを作成します。 AWS IoT GreengrassCore ソフトウェアログには、Greengrass nucleus とプラグインコンポーネントのログが含まれます。

```
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true"
    }
  }
}
```

Greengrass コンポーネントのログを CloudWatch Logs に書き込むようにコアデバイスを設定するには、コンポーネントをコンポーネントログ記録設定のリストに追加する設定更新を指定するデプロイを作成します。このリストにコンポーネントを追加すると、ログマネージャーコンポーネントは CloudWatch ログにログを書き込みます。コンポーネントログには、ジェネリックコンポーネントと Lambda コンポーネントのログが含まれます。

```
{
  "logsUploaderConfiguration": {
    "componentLogsConfigurationMap": {
      "com.example.HelloWorld": {

      }
    }
  }
}
```

ログマネージャーコンポーネントをデプロイするときに、ディスク容量の制限と、コアデバイスが CloudWatch Logs に書き込んだ後にログファイルを削除するかどうかを設定することもできます。詳細については、「AWS IoT Greengrass のログ記録の設定」を参照してください。

AWS IoT Greengrass のログ記録の設定

次のオプションを設定して Greengrass コアデバイスのログをカスタマイズできます。これらのオプションを設定するには、Greengrass nucleus またはログマネージャーコンポーネントへの設定更新を指定するデプロイの作成を行います。

- ログへの CloudWatch ログの書き込み

  コアデバイスをリモートでトラブルシューティングするには、 Core ソフトウェアとコンポーネントログを CloudWatch Logs に書き込むようにAWS IoT Greengrassコアデバイスを設定できます。これを行うには、ログマネージャーコンポーネントをデプロイして設定します。詳細については、「 CloudWatch ログへのログ記録を有効にする」を参照してください。
- アップロードされたログファイルの削除

  ディスク容量の使用量を減らすために、ログファイルを CloudWatch Logs に書き込んだ後にログファイルを削除するようにコアデバイスを設定できます。詳細については、ログマネージャーコンポーネントの `deleteLogFileAfterCloudUpload` パラメータを参照してください。これは、AWS IoT Greengrass Core ソフトウェアログとコンポーネントログを対象として指定できます。
- ログディスクの容量制限

  ディスク容量の使用量を制限するには、コアデバイスの各ログに対して最大ディスク容量 (ローテーションされたログファイルを含む) を設定できます。例えば、`greengrass.log` とローテーションされた `greengrass.log` ファイルの最大合計ディスク容量を設定できます。詳細については、Greengrass nucleus コンポーネントの `logging.totalLogsSizeKB` パラメータとログマネージャーコンポーネントの `diskSpaceLimit` パラメータを参照してください。これは、AWS IoT Greengrass Core ソフトウェアログとコンポーネントログを対象として指定できます。
- ログファイルのサイズ制限

  各ログファイルの最大ファイルサイズを設定できます。ログファイルがこのファイルサイズの上限を超えると、AWS IoT Greengrass Core ソフトウェアは新しいログファイルを作成します。ログマネージャーコンポーネントバージョン 2.28 (およびそれ以前) では、ローテーションされたログファイルのみが CloudWatch Logs に書き込まれるため、より低いファイルサイズの制限を指定して、ログに CloudWatch ログをより頻繁に書き込むことができます。ログマネージャーコンポー

ネントバージョン 2.3.0 (およびそれ以降) は、ローテーションを待たずにすべてのログを処理してアップロードします。詳細については、Greengrass nucleus コンポーネントのログファイルサイズの上限パラメータ (`logging.fileSizeKB`) を参照してください。

- 最小ログレベル

  Greengrass nucleus コンポーネントがファイル システム ログに書き込む最小ログレベルを設定できます。例えば、トラブルシューティングを支援するために `DEBUG` レベルのログを指定、あるいはコアデバイスが作成するログの量を減らすために `ERROR` レベルログを指定できます。詳細については、Greengrass nucleus コンポーネントのログレベルパラメータ (`logging.level`) を参照してください。

  ログマネージャーコンポーネントが CloudWatch ログに書き込む最小ログレベルを設定することもできます。例えば、より高いログレベルを指定してログコストを減らすこともできます。詳細については、ログマネージャーコンポーネントの `minimumLogLevel` パラメータを参照してください。これは、AWS IoT Greengrass Core ソフトウェアログとコンポーネントログを対象として指定できます。

- ログに書き込む CloudWatch ログをチェックする間隔

  ログマネージャーコンポーネントが CloudWatch ログにログを書き込む頻度を増減するには、新しいログファイルが書き込まれることを確認する間隔を設定できます。例えば、デフォルトの 5 分間隔よりも早く CloudWatch Logs のログを表示するように、間隔を短く指定できます。ログマネージャーコンポーネントはログファイルをより少ないリクエストにバッチ処理するので、コストを抑えるために間隔を長く指定できます。詳細については、ログマネージャーコンポーネントのアップロード間隔パラメータ (`periodicUploadIntervalSec`) を参照してください。

- ログ形式

  AWS IoT Greengrass Core ソフトウェアがログをテキストまたは JSON 形式のいずれかに書き込むように選択できます。ログを読み取る場合、テキスト形式を選択します。または、アプリケーションを使用してログの読み取りまたはパーシングする場合、JSON 形式を選択します。詳細については、Greengrass nucleus コンポーネントのログ形式パラメータ (`logging.format`) を参照してください。

- ローカルファイルシステムのログフォルダ

  ログフォルダを */greengrass/v2*/logs からコアデバイスの別なフォルダに変更できます。詳細については、Greengrass nucleus コンポーネントの出力ディレクトリパラメータ (`logging.outputDirectory`) を参照してください。

AWS CloudTrail ログ

AWS IoT Greengrass は、ユーザー、ロール、AWS IoT Greengrass の AWS のサービス が行ったアクションの記録を提供するサービスである AWS CloudTrail と統合されています。詳細については、「を使用した AWS IoT Greengrass V2 API コールのログ記録 AWS CloudTrail」を参照してください。

を使用した AWS IoT Greengrass V2 API コールのログ記録 AWS CloudTrail

AWS IoT Greengrass V2 は、 のユーザー AWS CloudTrail、ロール、または AWS のサービスによって実行されたアクションを記録するサービスである と統合されています AWS IoT Greengrass Version 2。 は、 のすべての API コールをイベント AWS IoT Greengrass として CloudTrail キャプチャします。キャプチャされる呼び出しには、 AWS IoT Greengrass コンソールからの呼び出しと AWS IoT Greengrass API オペレーションへのコード呼び出しが含まれます。

証跡を作成する場合は、 の CloudTrail イベントなど、S3 バケットへのイベントの継続的な配信を有効にすることができます AWS IoT Greengrass。証跡を設定しない場合でも、 CloudTrail コンソールのイベント履歴 で最新のイベントを表示できます。で収集された情報を使用して CloudTrail、 に対するリクエスト AWS IoT Greengrass、リクエスト元の IP アドレス、リクエスト者、リクエスト日時などの詳細を確認できます。

の詳細については CloudTrail、「 AWS CloudTrail ユーザーガイド」を参照してください。

AWS IoT Greengrass V2 内の情報 CloudTrail

CloudTrail アカウントを作成する AWS アカウント と、 は で有効になります。でアクティビティが発生すると AWS IoT Greengrass、そのアクティビティは CloudTrail イベント履歴 の他の AWS サービスイベントとともに イベントに記録されます。 最近のイベントは、 AWS アカウントで表示、検索、ダウンロードできます。詳細については、「イベント履歴 を使用した CloudTrail イベントの表示」を参照してください。

のイベントなど AWS アカウント、 のイベントの継続的な記録については AWS IoT Greengrass、証跡を作成します。証跡により、 はログファイル CloudTrail を S3 バケットに配信できます。デフォルトでは、コンソールで証跡を作成すると、証跡はすべての に適用されます AWS リージョン。証跡は、 AWS パーティション内のすべてのリージョンからのイベントを口グに記録し、指定した S3 バケットにログファイルを配信します。さらに、 CloudTrail ログで収集されたデータをより詳細に

分析し、それに基づく対応を行うように他の AWS サービスを設定できます。詳細については、次を参照してください:

- 「追跡の作成の概要」
- CloudTrail でサポートされているサービスと統合
- の Amazon SNS 通知の設定 CloudTrail
- 複数のリージョンからの CloudTrail ログファイルの受信と複数のアカウントからの CloudTrail ログファイルの受信

すべての AWS IoT Greengrass V2 アクションは によって ログに記録 CloudTrail され、AWS IoT Greengrass V2 API リファレンス に記載されています。例えば、、、および `CancelDeployment`アクションを呼び出す`CreateDeployment`と`CreateComponentVersion`、CloudTrail ログファイルにエントリが生成されます。

各イベントまたはログエントリには、誰がリクエストを生成したかという情報が含まれます。アイデンティティ情報は、以下を判別するために役立ちます。

- リクエストがルートまたは AWS Identity and Access Management (IAM) ユーザーの認証情報のどちらを使用して送信されたか。
- リクエストがロールまたはフェデレーションユーザーのテンポラリなセキュリティ認証情報を使用して行われたかどうか。
- リクエストが別の AWS サービスによって送信されたかどうか。

詳細については、「CloudTrail userIdentity 要素」を参照してください。

AWS IoT Greengrass での データイベント CloudTrail

データイベントは、リソースで、またはリソースで実行されたリソースオペレーションに関する情報を提供します (コンポーネントバージョンやデプロイの設定の取得など)。これらのイベントは、データプレーンオペレーションとも呼ばれます。データイベントは、多くの場合、高ボリュームのアクティビティです。デフォルトでは、 CloudTrail はデータイベントを記録しません。 CloudTrail イベント履歴にはデータイベントは記録されません。

追加の変更がイベントデータに適用されます。 CloudTrail 料金の詳細については、AWS CloudTrail 「 の料金」を参照してください。

CloudTrail コンソール、、または CloudTrail API オペレーションを使用して AWS CLI、 AWS IoT Greengrass リソースタイプのデータイベントをログに記録できます。このセクションの表は、 で使用できるリソースタイプを示しています AWS IoT Greengrass。

- CloudTrail コンソールを使用してデータイベントを記録するには、証跡またはイベントデータストアを作成してデータイベントを記録するか、既存の証跡またはイベントデータストアを更新してデータイベントをログに記録します。

  1. データイベントを選択して、データイベントをログに記録します。
  2. データイベントタイプのリストから、データイベントを記録するリソースタイプを選択します。
  3. 使用するログセレクタテンプレートを選択します。リソースタイプのすべてのデータイベントをログに記録したり、すべての`readOnly`イベントをログに記録したり、すべての`writeOnly`イベントをログに記録したり、カスタムログセレクタテンプレートを作成して`readOnly`、、`eventName`および `resources.ARN`フィールドでフィルタリングしたりできます。

- を使用してデータイベントをログに記録するには AWS CLI、 `eventCategory`フィールドを `Data`に、 `resources.type`フィールドをリソースタイプ値に等しく設定するように `--advanced-event-selectors`パラメータを設定します (表 を参照)。、`readOnly`、`eventName`および `resources.ARN`フィールドの値でフィルタリングする条件を追加できます。
  - データイベントを記録するように証跡を設定するには、 put-event-selectors コマンドを実行します。詳細については、「 を使用した証跡のデータイベントの口グ記録 AWS CLI」を参照してください。
  - データイベントをログに記録するようにイベントデータストアを設定するには、 create-event-data-store コマンドを実行してデータイベントをログに記録する新しいイベントデータストアを作成するか、 update-event-data-store コマンドを実行して既存のイベントデータストアを更新します。詳細については、「 を使用したイベントデータストアのデータイベントの口グ記録 AWS CLI」を参照してください。

次の表に、 AWS IoT Greengrass リソースタイプを示します。データイベントタイプ (コンソール) 列には、 CloudTrail コンソールのデータイベントタイプリストから選択する値が表示されます。resources.type 値列には、 AWS CLI または CloudTrail APIsを使用して高度なイベントセレクタを設定するときに指定する `resources.type`値が表示されます。列に口グ記録された Data APIs CloudTrail には、リソースタイプについて に CloudTrail口グ記録された API コールが表示されます。

データイベントタイプ (コンソール)	resources.type 値	にログ記録APIs CloudTrail
IoT 証明書	`AWS::IoT::Certificate`	• VerifyClientDeviceIdentity • VerifyClientDeviceIoTCertificateAssociation
IoT Greengrass コンポーネントバージョン	`AWS::GreengrassV2::ComponentVersion`	• ResolveComponentCandidates
IoT Greengrass デプロイ	`AWS::GreengrassV2::Deployment`	• GetDeploymentConfiguration
IoT モノ	`AWS::IoT::Thing`	• ListThingGroupsForCoreDevices • PutCertificateAuthorities • VerifyClientDeviceIoTCertificateAssociation

> **Note**
>
> Greengrass はアクセス拒否イベントを記録しません。

`eventName`、`readOnly`、および `resources.ARN` フィールドでフィルタリングして、自分にとって重要なイベントのみをログに記録するように高度なイベントセレクタを設定できます。

にフィルターを追加して`eventName`、特定のデータ APIs。

フィールドの詳細については、「AdvancedFieldSelector」を参照してください。

次の例は、 を使用して高度なセレクタを設定する方法を示しています AWS CLI。*`TrailName`* と *`region`* をユーザー自身の情報に置き換えます。

Example — IoT モノのデータイベントを log に記録する

```
aws cloudtrail put-event-selectors --trail-name TrailName --region region \
--advanced-event-selectors \
```

```
‘[
    {
            “Name”: “Log all thing data events”,
            “FieldSelectors”: [
                { “Field”: “eventCategory”, “Equals”: [“Data”] },
                { “Field”: “resources.type”, “Equals”: [“AWS::IoT::Thing”] }
            ]
        }
]’
```

Example - 特定の IoT モノの API でフィルタリングする

```
aws cloudtrail put-event-selectors --trail-name TrailName --region region \
--advanced-event-selectors \
‘[
    {
            “Name”: “Log IoT Greengrass PutCertificateAuthorities API calls”,
            “FieldSelectors”: [
                { “Field”: “eventCategory”, “Equals”: [“Data”] },
                { “Field”: “resources.type”, “Equals”: [“AWS::IoT::Thing”] },
                { “Field”: “eventName”, “Equals”: [“PutCertificateAuthorities”] }
            ]
    }
]’
```

Example – すべての Greengrass データイベントを口グに記録する

```
aws cloudtrail put-event-selectors --trail-name TrailName --region region \
--advanced-event-selectors \
‘[
    {
        “Name”: “Log all certificate data events”,
        “FieldSelectors”: [
            {
                “Field”: “eventCategory”,
                “Equals”: [
                    “Data”
                ]
            },
            {
                “Field”: “resources.type”,
                “Equals”: [
                    “AWS::IoT::Certificate”
```

```
                ]
            }
        ]
    },
    {
        “Name”: “Log all component version data events”,
        “FieldSelectors”: [
            {
                “Field”: “eventCategory”,
                “Equals”: [
                    “Data”
                ]
            },
            {
                “Field”: “resources.type”,
                “Equals”: [
                    “AWS::GreengrassV2::ComponentVersion”
                ]
            }
        ]
    },
    {
        “Name”: “Log all deployment version”,
        “FieldSelectors”: [
            {
                “Field”: “eventCategory”,
                “Equals”: [
                    “Data”
                ]
            },
            {
                “Field”: “resources.type”,
                “Equals”: [
                    “AWS::GreengrassV2::Deployment”
                ]
            }
        ]
    },
    {
        “Name”: “Log all thing data events”,
        “FieldSelectors”: [
            {
                “Field”: “eventCategory”,
                “Equals”: [
```

```
                    “Data”
                ]
            },
            {
                “Field”: “resources.type”,
                “Equals”: [
                    “AWS::IoT::Thing”
                ]
            }
        ]
    }
]’
```

AWS IoT Greengrass での 管理イベント CloudTrail

管理イベントは、 AWS アカウントのリソースで実行される管理オペレーションに関する情報を提供します。これらのイベントは、コントロールプレーンオペレーションとも呼ばれます。デフォルトでは、 は管理イベント CloudTrail を記録します。

AWS IoT Greengrass は、すべての AWS IoT Greengrass コントロールプレーンオペレーションを管理イベントとして記録します。が AWS IoT Greengrass に記録する AWS IoT Greengrass コントロールプレーンオペレーションのリストについては CloudTrail、 AWS IoT Greengrass API リファレンス、バージョン 2 を参照してください。

AWS IoT Greengrass V2 ログファイルエントリについて

証跡は、指定した S3 バケットにイベントをログファイルとして配信できるようにする設定です。CloudTrail ログファイルには、1 つ以上のログエントリが含まれます。イベント は、任意の送信元からの単一の要求を表します。これには、リクエストされたアクション、アクションの日時、リクエストパラメータなどに関する情報が含まれます。 CloudTrail ログファイルは、パブリック API コールの順序付けられたスタックトレースではないため、特定の順序では表示されません。

次の例は、 `CreateDeployment`アクションを示す CloudTrail ログエントリを示しています。

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/Administrator",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
```

```
        "userName": "Administrator"
    },
    "eventTime": "2021-01-06T02:38:05Z",
    "eventSource": "greengrass.amazonaws.com",
    "eventName": "CreateDeployment",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "203.0.113.0",
    "userAgent": "aws-cli/2.1.9 Python/3.7.9 Windows/10 exe/AMD64 prompt/off command/
greengrassv2.create-deployment",
    "requestParameters": {
        "deploymentPolicies": {
            "failureHandlingPolicy": "DO_NOTHING",
            "componentUpdatePolicy": {
                "timeoutInSeconds": 60,
                "action": "NOTIFY_COMPONENTS"
            },
            "configurationValidationPolicy": {
                "timeoutInSeconds": 60
            }
        },
        "deploymentName": "Deployment for MyGreengrassCoreGroup",
        "components": {
            "aws.greengrass.Cli": {
                "componentVersion": "2.0.3"
            }
        },
        "iotJobConfiguration": {},
        "targetArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/
MyGreengrassCoreGroup"
    },
    "responseElements": {
        "iotJobArn": "arn:aws:iot:us-west-2:123456789012:job/fdfeba1d-ac6d-44ef-
ab28-54f684ea578d",
        "iotJobId": "fdfeba1d-ac6d-44ef-ab28-54f684ea578d",
        "deploymentId": "4196dddc-0a21-4c54-a985-66a525f6946e"
    },
    "requestID": "311b9529-4aad-42ac-8408-c06c6fec79a9",
    "eventID": "c0f3aa2c-af22-48c1-8161-bad4a2ab1841",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "eventCategory": "Management",
    "recipientAccountId": "123456789012"
```

```
}
```

AWS IoT Greengrass コアデバイスからシステムヘルステレメトリデータを収集する

システムヘルステレメトリデータは、Greengrass コアデバイスで重要な操作のパフォーマンス監視に役立つ診断データです。プロジェクトとアプリケーションを作成して、エッジデバイスからのテレメトリデータを取得、分析、変換、レポートできます。プロセスエンジニアといった特定分野のエキスパートは、これらのアプリケーションを使用して、フリートのヘルスに関する洞察を得られます。

次の方法を使用して Greengrass コアデバイスからテレメトリデータを収集できます。

- Nucleus テレメトリエミッタコンポーネント - [nucleus telemetry emitter component] (nucleus テレメトリエミッタコンポーネント)(`aws.greengrass.telemetry.NucleusEmitter`) は、デフォルトでテレメトリデータを `$local/greengrass/telemetry` トピックに公開します。このトピックに公開されるデータを使用して、デバイスのクラウドへの接続が制限されている場合でも、コアデバイスのローカルで動作できます。オプションで、テレメトリデータを選択した AWS IoT Core MQTT トピックに公開するようにコンポーネントを設定することもできます。

  テレメトリデータを公開するには、nucleus エミッタコンポーネントをコアデバイスにデプロイする必要があります。テレメトリデータをローカルトピックに公開することに関連するコストはありません。ただし、データを AWS クラウド に公開するための MQTT トピックの使用には、[AWS IoT Core pricing] (価格設定) が適用されます。

  AWS IoT Greengrass はいくつかのコミュニティコンポーネントを提供し、InfluxDB と Grafana を使用してコアデバイスでローカルにテレメトリデータを分析と可視化するうえで役立ちます。これらのコンポーネントは、nucleus エミッタコンポーネントのテレメトリデータを使用します。詳細については、InfluxDB パブリッシャーコンポーネントの「README」を参照してください。
- テレメトリエージェント — Greengrass コアデバイスのテレメトリエージェントは、ローカルテレメトリデータを収集し、お客様とのやり取りを必要と EventBridge せずに Amazon に公開します。コアデバイスは、ベストエフォートベース EventBridge でテレメトリデータを に発行します。例えば、コアデバイスはオフライン中にテレメトリデータの配信に失敗することがあります。

  テレメトリエージェント機能は、すべての Greengrass コアデバイスのデフォルトで有効になっています。Greengrass コアデバイスをセットアップすると、すぐに自動的にデータの受信が開始されます。データリンクのコストを除き、コアデバイスから AWS IoT Core へのデータ転送には料金は発生しません。これは、エージェントが AWS の予約済みトピックに公開しているためです。た

だし、ユースケースによっては、データを受信または処理するときにコストが発生する場合もあります。

> **Note**
>
> Amazon EventBridge は、アプリケーションを Greengrass コアデバイスなどのさまざまなソースのデータに接続するために使用できるイベントバスサービスです。詳細については、「Amazon ユーザーガイド」の「Amazon EventBridgeとは」を参照してください。EventBridge

AWS IoT Greengrass Core ソフトウェアを適切に機能させるために、AWS IoT Greengrass は開発および品質改善の目的でデータを使用します。この機能は、エッジ機能や拡張エッジ機能にも役立ちます。AWS IoT Greengrass は、テレメトリデータを最大 7 日間保持します。

このセクションでは、テレメトリーエージェントを設定して使用する方法について説明します。nucleus テレメトリエミッタコンポーネントの設定については、「nucleus テレメトリエミッタ」を参照してください。

トピック

- テレメトリメトリクス
- テレメトリエージェント設定を設定する
- でテレメトリデータをサブスクライブする EventBridge

テレメトリメトリクス

テレメトリエージェントによって公開されるメトリクスの説明を次の表に示します。

名前	説明	
システム		
`SystemMemUsage`	オペレーティングシステムを含む、Greengrass コアデバイスのすべてのアプリケーションで現在使用されているメモリの量。	

名前	説明
CpuUsage	オペレーティングシステムを含む Greengrass コアデバイスのすべてのアプリケーションで現在使用されている CPU の量。
TotalNumberOfFDs	Greengrass コアデバイスのオペレーティングシステムによって保存されているファイルディスクリプタの数。1 つのファイルディスクリプタは、1 つのオープンファイルを一意に識別します。
Greengrass nucleus	
NumberOfComponentsRunning	Greengrass コアデバイスで実行されているコンポーネント数。
NumberOfComponentsErrored	Greengrass コアデバイスでエラー状態にあるコンポーネントの数。
NumberOfComponentsInstalled	Greengrass コアデバイスでインストールされているコンポーネントの数。
NumberOfComponentsStarting	Greengrass コアデバイスで開始されているコンポーネントの数。
NumberOfComponentsNew	Greengrass コアデバイスで新しくなったコンポーネントの数。

名前	説明	
NumberOfComponents Stopping	Greengrass コアデバイスで停止しているコンポーネントの数。	
NumberOfComponents Finished	Greengrass コアデバイスで終了するコンポーネントの数。	
NumberOfComponents Broken	Greengrass コアデバイスで壊れているコンポーネントの数。	
NumberOfComponents Stateless	Greengrass コアデバイスでステートレスであるコンポーネントの数。	
クライアントデバイス認証 – この機能には、v2.4.0 以降のクライアントデバイス認証コンポーネントが必要です。		
VerifyClientDevice Identity.Success	クライアントデバイスの ID の検証に成功した回数。	
VerifyClientDevice Identity.Failure	クライアントデバイスの ID の検証に失敗した回数。	
AuthorizeClientDev iceActions.Success	クライアントデバイスが要求されたアクションを完了することが許可された回数。	
AuthorizeClientDev iceActions.Failure	クライアントデバイスが要求されたアクションを完了することが許可されない回数。	
GetClientDeviceAut hToken.Success	クライアント端末の認証に成功した回数。	

名前	説明	
GetClientDeviceAuthToken.Failure	クライアント端末の認証に失敗した回数。	
SubscribeToCertificateUpdates.Success	証明書の更新に成功したサブスクリプションの数。	
SubscribeToCertificateUpdates.Failure	証明書の更新のサブスクライブしようとして失敗した回数。	
ServiceError	クライアントデバイスの auth 全体で、処理されなかった内部エラーの数。	
ストリームマネージャー – この機能を使用するには、Greengrass nucleus コンポーネントの v2.7.0 以降が必要です。		
BytesAppended	ストリームマネージャーに追加されたデータのバイト数。	
BytesUploadedToIoTAnalytics	ストリームマネージャーが AWS IoT Analytics のチャネルにエクスポートするデータのバイト数。	
BytesUploadedToKinesis	ストリームマネージャーが Amazon Kinesis Data Streams のストリームにエクスポートするデータのバイト数。	

名前	説明	
BytesUploadedToIoTSiteWise	ストリームマネージャーが AWS IoT SiteWise のアセットプロパティにエクスポートするデータのバイト数。	
BytesUploadedToS3	ストリームマネージャーが Amazon S3 のオブジェクトにエクスポートするデータのバイト数。	

テレメトリエージェント設定を設定する

テレメトリエージェントは、次のデフォルト設定を使用します。

- テレメトリエージェントは、1 時間ごとにテレメトリデータを集約します。
- テレメトリエージェントは 24 時間ごとにテレメトリメッセージを発行します。

テレメトリエージェントは、サービス品質 (QoS) レベルが 0 の MQTT プロトコルを使用してデータを公開します。つまり、配信の確認や公開の再試行は行われません。テレメトリメッセージは、MQTT 接続を、AWS IoT Core を送信先とする他のサブスクリプションメッセージと共有します。

データリンクのコストを除き、コアから AWS IoT Core へのデータ転送には料金は発生しません。これは、エージェントが AWS の予約済みトピックに公開しているためです。ただし、ユースケースによっては、データを受信または処理するときにコストが発生する場合もあります。

Greengrass コアデバイスごとに、テレメトリエージェント機能を有効または無効にできます。コアデバイスがデータを集約して公開する間隔を設定することもできます。テレメトリを設定するには、Greengrass nucleus コンポーネントをデプロイするときに、テレメトリ設定パラメータをカスタマイズします。

でテレメトリデータをサブスクライブする EventBridge

Amazon でルールを作成して EventBridge 、Greengrass コアデバイスのテレメトリエージェントから発行されたテレメトリデータの処理方法を定義できます。がデータ EventBridge を受信すると、

ルールで定義されているターゲットアクションが呼び出されます。例えば、通知の送信、イベント情報の保存、是正措置の実践、他のイベントの呼び出しなどを行うイベントルールを作成できます。

テレメトリイベント

テレメトリイベントは次の形式を使用します。

```
{
  "version": "0",
  "id": "a09d303e-2f6e-3d3c-a693-8e33f4fe3955",
  "detail-type": "Greengrass Telemetry Data",
  "source": "aws.greengrass",
  "account": "123456789012",
  "time": "2020-11-30T20:45:53Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
    "ThingName": "MyGreengrassCore",
    "Schema": "2020-07-30",
    "ADP": [
      {
        "TS": 1602186483234,
        "NS": "SystemMetrics",
        "M": [
          {
            "N": "TotalNumberOfFDs",
            "Sum": 6447.0,
            "U": "Count"
          },
          {
            "N": "CpuUsage",
            "Sum": 15.458333333333332,
            "U": "Percent"
          },
          {
            "N": "SystemMemUsage",
            "Sum": 10201.0,
            "U": "Megabytes"
          }
        ]
      },
      {
        "TS": 1602186483234,
```

```
        "NS": "GreengrassComponents",
        "M": [
          {
            "N": "NumberOfComponentsStopping",
            "Sum": 0.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsStarting",
            "Sum": 0.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsBroken",
            "Sum": 0.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsFinished",
            "Sum": 1.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsInstalled",
            "Sum": 0.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsRunning",
            "Sum": 7.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsNew",
            "Sum": 0.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsErrored",
            "Sum": 0.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsStateless",
```

```
            "Sum": 0.0,
            "U": "Count"
          }
        ]
      },
      {
        "TS": 1602186483234,
        "NS": "aws.greengrass.ClientDeviceAuth",
        "M": [
          {
            "N": "VerifyClientDeviceIdentity.Success",
            "Sum": 3.0,
            "U": "Count"
          },
          {
            "N": "VerifyClientDeviceIdentity.Failure",
            "Sum": 1.0,
            "U": "Count"
          },
          {
            "N": "AuthorizeClientDeviceActions.Success",
            "Sum": 20.0,
            "U": "Count"
          },
          {
            "N": "AuthorizeClientDeviceActions.Failure",
            "Sum": 5.0,
            "U": "Count"
          },
          {
            "N": "GetClientDeviceAuthToken.Success",
            "Sum": 5.0,
            "U": "Count"
          },
          {
            "N": "GetClientDeviceAuthToken.Failure",
            "Sum": 2.0,
            "U": "Count"
          },
          {
            "N": "SubscribeToCertificateUpdates.Success",
            "Sum": 10.0,
            "U": "Count"
          },
```

```
          {
            "N": "SubscribeToCertificateUpdates.Failure",
            "Sum": 1.0,
            "U": "Count"
          },
          {
            "N": "ServiceError",
            "Sum": 3.0,
            "U": "Count"
          }
        ]
      },
      {
        "TS": 1602186483234,
        "NS": "aws.greengrass.StreamManager",
        "M": [
          {
            "N": "BytesAppended",
            "Sum": 157745524.0,
            "U": "Bytes"
          },
          {
            "N": "BytesUploadedToIoTAnalytics",
            "Sum": 149012.0,
            "U": "Bytes"
          },
          {
            "N": "BytesUploadedToKinesis",
            "Sum": 12192.0,
            "U": "Bytes"
          },
          {
            "N": "BytesUploadedToIoTSiteWise",
            "Sum": 13321.0,
            "U": "Bytes"
          },
          {
            "N": "BytesUploadedToS3",
            "Sum": 12213.0,
            "U": "Bytes"
          }
        ]
      }
    ]
```

```
  }
}
```

ADP 配列には、次のプロパティを持つ集約データポイントのリストが含まれています。

TS

データが収集された時刻のタイムスタンプ。

NS

メトリクスの名前空間。

M

メトリクスのリスト。メトリクスには次のプロパティが含まれています。

N

メトリクスの名前。

Sum

このテレメトリイベントのメトリクス値の合計。

U

メトリクス値の単位。

各メトリクスの詳細については、「テレメトリメトリクス」を参照してください。

ルールを作成 EventBridgeするための前提条件

の EventBridge ルールを作成する前にAWS IoT Greengrass、以下を実行する必要があります。

- のイベント、ルール、ターゲットをよく理解してください EventBridge。
- EventBridge ルールによって呼び出されるターゲットを作成して設定します。ルールによって、Amazon Kinesis ストリーム、AWS Lambda 関数、Amazon SNS トピック、Amazon SQS キューなど、さまざまなタイプのターゲットを呼び出すことができます。

  EventBridge ルールと関連するターゲットは、Greengrass リソースを作成した AWS リージョンにある必要があります。詳細については、「AWS 全般のリファレンス」の「サービスエンドポイントとクォータ」を参照してください。

詳細については、「Amazon ユーザーガイド」の「Amazon EventBridgeとは」および「Amazon の開始 EventBridge方法」を参照してください。 EventBridge

テレメトリデータを取得するイベントルールを作成する (コンソール)

を使用して、Greengrass コアデバイスによって発行されたテレメトリデータを受信する EventBridge ルールAWS Management Consoleを作成するには、次のステップを実行します。これにより、ウェブサーバー、E メールアドレス、その他のトピック受信者がイベントに応答できるようになります。詳細については、「Amazon EventBridge ユーザーガイド」の「 AWSリソースからのイベントでトリガーする EventBridge ルールの作成」を参照してください。

1. Amazon EventBridge コンソール を開き、ルールの作成 を選択します。
2. [Name and description (名前と説明)] に、ルールの名前と説明を入力します。
3. [Define pattern (パターンの定義)] で、ルールパターンを設定します。
   a. [イベントパターン] を選択します。
   b. [Pre-defined pattern by service (サービスによる定義済みパターン)] を選択します。
   c. Service provider (サービスプロバイダー)で、AWSを選択します。
   d. [Service name (サービス名)] で [Greengrass] を選択します。
   e. [Event type] (イベントタイプ) では、[Greengrass Telemetry Data] (Greengrass テレメトリデータ) を選択します。
4. [Select event bus (イベントバスの選択)] では、イベントバスのオプションはデフォルトのままにしておきます。
5. [Select targets (ターゲットの選択)] で、ターゲットを設定します。次の例では、Amazon SQS キューを使用していますが、他のターゲットタイプも設定できます。
   a. [Target] (ターゲット) で、[SQS queue] (SQS キュー) を選択します。
   b. [Queue**] (キュー**) でターゲットキューを選択します。
6. [Tags - optional (タグ (オプション))] で、ルールのタグを定義するか、フィールドを空のままにします。
7. [作成] を選択します。

テレメトリデータを取得するイベントルールを作成する (CLI)

を使用して、Greengrass コアデバイスによって発行されたテレメトリデータを受信する EventBridge ルールAWS CLIを作成するには、次のステップを実行します。これにより、ウェブサーバー、E メールアドレス、その他のトピック受信者がイベントに応答できるようになります。

1. ルールを作成します。

   - *thing-name* をコアデバイスのモノ名に置き換えます。

   Linux or Unix

   ```
   aws events put-rule \
     --name MyGreengrassTelemetryEventRule \
     --event-pattern "{\"source\": [\"aws.greengrass\"], \"detail\": {\"ThingName
   \": [\"thing-name\"]}}"
   ```

   Windows Command Prompt (CMD)

   ```
   aws events put-rule ^
     --name MyGreengrassTelemetryEventRule ^
     --event-pattern "{\"source\": [\"aws.greengrass\"], \"detail\": {\"ThingName
   \": [\"thing-name\"]}}"
   ```

   PowerShell

   ```
   aws events put-rule `
     --name MyGreengrassTelemetryEventRule `
     --event-pattern "{\"source\": [\"aws.greengrass\"], \"detail\": {\"ThingName
   \": [\"thing-name\"]}}"
   ```

   パターンで省略されたプロパティは無視されます。

2. トピックをルールターゲットとして追加します。次の例では、Amazon SQS を使用していますが、他のターゲットタイプも設定できます。

   - *queue-arn* を Amazon SQS キューの ARN に置き換えます。

Linux or Unix

```
aws events put-targets \
  --rule MyGreengrassTelemetryEventRule \
  --targets "Id"="1","Arn"="queue-arn"
```

Windows Command Prompt (CMD)

```
aws events put-targets ^
  --rule MyGreengrassTelemetryEventRule ^
  --targets "Id"="1","Arn"="queue-arn"
```

PowerShell

```
aws events put-targets `
  --rule MyGreengrassTelemetryEventRule `
  --targets "Id"="1","Arn"="queue-arn"
```

> **Note**
>
> Amazon EventBridge がターゲットキューを呼び出せるようにするには、トピックにリソースベースのポリシーを追加する必要があります。詳細については、「Amazon ユーザーガイド」の「Amazon SQS アクセス許可 EventBridge 」を参照してください。

詳細については、「Amazon ユーザーガイド」の「 のイベントとイベントパターン EventBridge」を参照してください。 EventBridge

デプロイとコンポーネントのヘルスステータス通知を受け取る

Amazon EventBridge イベントルールは、デバイスが受信した Greengrass デプロイとデバイスにインストールされたコンポーネントの状態変更に関する通知を提供します。 EventBridge は、 AWSリソースの変更を示すシステムイベントのほぼリアルタイムのストリームを提供します。 は、これらのイベントをベストエフォートベース EventBridge で AWS IoT Greengrassに送信します。つまり、はすべてのイベントを に送信AWS IoT Greengrassしようとします EventBridge が、まれにイベントが配信されない場合があります。さらに、AWS IoT Greengrass は特定のイベントの複数のコピーを

送信する場合があります。これは、イベントリスナーがイベントの発生順にイベントを受信しない可能性があることを意味します。

> **Note**
>
> Amazon EventBridge は、アプリケーションを Greengrass コアデバイスやデプロイ、コンポーネント通知などのさまざまなソースからのデータに接続するために使用できるイベントバスサービスです。詳細については、「Amazon ユーザーガイド」の「Amazon EventBridge とは」を参照してください。 EventBridge

トピック

デプロイステータスの変更イベント

デプロイが、AWS IoT Greengrass、`FAILED`、および `COMPLETED` 状態になると、`SUCCEEDED` はイベントを発行します。すべての状態遷移または指定した状態への移行に対して実行される EventBridge ルールを作成できます。デプロイがルールを開始する状態になると、 はルールで定義されたターゲットアクションを EventBridge 呼び出します。これにより、通知を送信したり、イベント情報をキャプチャしたり、修正アクションを実行したり、状態の変更に応じて他のイベントを開始したりできます。例えば、次のユースケースのルールを作成できます。

- アセットのダウンロードや担当者の通知など、デプロイ後のオペレーションを開始します。
- デプロイの成功または失敗時に通知を送信します。
- デプロイイベントに関するカスタムメトリクスを発行します。

デプロイ状態変更のイベントでは、次の形式を使用します。

```
{
    "version":"0",
    "id":" cd4d811e-ab12-322b-8255-EXAMPLEb1bc8",
    "detail-type":"Greengrass V2 Effective Deployment Status Change",
    "source":"aws.greengrass",
    "account":"123456789012",
    "region":"us-west-2",
    "time":"2018-03-22T00:38:11Z",
    "resources":["arn:aws:greengrass:us-
east-1:123456789012:coreDevices:MyGreengrassCore"],
    "detail":{
        "deploymentId": "4f38f1a7-3dd0-42a1-af48-EXAMPLE09681",
        "coreDeviceExecutionStatus": "FAILED|SUCCEEDED|COMPLETED",
        "statusDetails": {
          "errorStack": ["DEPLOYMENT_FAILURE", "ARTIFACT_DOWNLOAD_ERROR", "S3_ERROR",
 "S3_ACCESS_DENIED", "S3_HEAD_OBJECT_ACCESS_DENIED"],
          "errorTypes": ["DEPENDENCY_ERROR", "PERMISSION_ERROR"],
        },
        "reason": "S3_HEAD_OBJECT_ACCESS_DENIED: FAILED_NO_STATE_CHANGE: Failed to
 download artifact name: 's3://pentest27/nucleus/281/aws.greengrass.nucleus.zip' for
 component aws.greengrass.Nucleus-2.8.1, reason: S3 HeadObject returns 403 Access
 Denied. Ensure the IAM role associated with the core device has a policy granting
 s3:GetObject. null (Service: S3, Status Code: 403, Request ID: HR94ZNT2161DAR58,
 Extended Request ID: wTX4DDI+qigQt3uzwl9rlnQiYlBgwvPm/KJFWeFAn9t1mnGXTms/
luLCYANgq08RIH+x2H+hEKc=)"
     }
}
```

デプロイのステータスを更新するルールやイベントを作成できます。FAILED、SUCCEEDED、または COMPLETED のいずれかとしてデプロイが完了すると、イベントが開始されます。コアデバイスでのデプロイが失敗した場合、その理由を説明する詳細なレスポンスを受け取ります。デプロイエラーコードの詳細については、「詳細なデプロイエラーコード」を参照してください。

デプロイの状態

- FAILED。デプロイに失敗しました。
- SUCCEEDED。モノのグループを対象としたデプロイが正常に完了しました。
- COMPLETED。モノを対象としたデプロイが正常に完了しました。

イベントが重複したり、順序が順不同である可能性があります。イベントの順序を決定するには、time プロパティを使用します。

errorStacks および のエラーコードの完全なリストについてはerrorTypes、詳細なデプロイエラーコード「」および「」を参照してください詳細なコンポーネントのステータスコード。

コンポーネントのステータス変更イベント

AWS IoT Greengrass はコンポーネントが次の状態になったときにイベントを発行します: ERRORED および BROKEN。Greengrass はデプロイの完了時にもイベントを発行します。すべての状態遷移または指定した状態への移行に対して実行される EventBridge ルールを作成できます。インストールされたコンポーネントがルールを開始する状態になると、 はルールで定義されたターゲットアクションを EventBridge 呼び出します。これにより、通知を送信したり、イベント情報をキャプチャしたり、修正アクションを実行したり、状態の変更に応じて他のイベントを開始したりできます。

コンポーネントの状態変更のイベントでは、次の形式を使用します。

```
{
    "version":"0",
    "id":" cd4d811e-ab12-322b-8255-EXAMPLEb1bc8",
    "detail-type":"Greengrass V2 Installed Component Status Change",
    "source":"aws.greengrass",
    "account":"123456789012",
    "region":"us-west-2",
    "time":"2018-03-22T00:38:11Z",
    "resources":["arn:aws:greengrass:us-
east-1:123456789012:coreDevices:MyGreengrassCore"],
    "detail": {
       "components": [
          {
            "componentName": "MyComponent",
            "componentVersion": "1.0.0",
            "root": true,
            "lifecycleState": "ERRORED|BROKEN",
            "lifecycleStatusCodes": ["STARTUP_ERROR"],
            "lifecycleStateDetails": "An error occurred during startup. The startup
 script exited with code 1."
          }
        ]
    }
}
```

インストールされたコンポーネントのステータスを更新するルールやイベントを作成できます。イベントは、コンポーネントの状態がデバイス上で変更されたときに開始されます。コンポーネントに工

ラーや故障が発生した理由を説明する詳細なレスポンスを受け取ります。また、失敗の理由を示すステータスコードも表示されます。コンポーネントのステータスコードの詳細については、「詳細なコンポーネントのステータスコード」を参照してください。

EventBridge ルールを作成するための前提条件

の EventBridge ルールを作成する前にAWS IoT Greengrass、次の操作を行います。

- のイベント、ルール、ターゲットをよく理解してください EventBridge。
- EventBridge ルールによって呼び出されるターゲットを作成して設定します。ルールは、以下のようなさまざまなタイプのターゲットを呼び出すことができます。
  - Amazon Simple Notification Service (Amazon SNS)
  - AWS Lambda 関数
  - Amazon Kinesis Video Streams
  - Amazon Simple Queue Service Amazon SQSキュー

詳細については、「Amazon ユーザーガイド」の「Amazon EventBridgeとは」および「Amazon の開始 EventBridge方法」を参照してください。 EventBridge

デバイスヘルス通知を設定する (コンソール)

グループのデプロイ状態が変更されたときに Amazon SNS トピックを発行する EventBridge ルールを作成するには、次のステップを使用します。これにより、ウェブサーバー、E メールアドレス、その他のトピック受信者がイベントに応答できるようになります。詳細については、「Amazon EventBridge ユーザーガイド」の「 AWSリソースからのイベントでトリガーする EventBridge ルールの作成」を参照してください。

1. Amazon EventBridge コンソール を開きます。
2. ナビゲーションペインで Rules] (ルール) を選択します。
3. ルールの作成 を選択します。
4. ルールの名前と説明を入力します。

   ルールには、同じリージョン内および同じイベントバス上の別のルールと同じ名前を付けることはできません。
5. Event bus] (イベントバス) では、このルールに関連付けるイベントバスを選択します。このルールをアカウントからのイベントと一致させるには、AWS のデフォルトのイベントバスを選択し

ます。アカウントの AWS サービスがイベントを発行すると、常にアカウントのデフォルトのイベントバスに移動します。

6. ルールタイプ では、イベントパターンを持つルール] を選択します。
7. [Next] (次へ) を選択します。
8. [Event source] (イベントソース) で、[AWSevents] (イベント) を選択します。
9. [イベントパターン] で、[AWS のサービス] を選択します。
10. [AWS のサービス] で [Greengrass] を選択します。
11. [Event type] (イベントタイプ) で、次から選択します。

    - デプロイイベントについては、[Greengrass V2 Effective Deployment Status Change] (Greengrass V2 の有効なデプロイステータスの変更) を選択します。
    - コンポーネントイベントについては、[Greengrass V2 Installed Component Status Change] (Greengrass V2 のインストールされたコンポーネントのステータス変更) を選択します。
12. 次へ をクリックします。
13. ターゲットタイプ] では、AWSサービス] を選択します。
14. [Select targets] (ターゲットの選択) で、ターゲットを設定します。この例では Amazon SNS トピックを使用していますが、通知を送信するターゲットタイプには他のトピックも設定できます。

    a. [Target (ターゲット)] で [SNS topic (SNS トピック)] を選択します。
    b. [Topic (トピック)] で、ターゲットトピックを選択します。
    c. [次へ] をクリックします。
15. 次へ をクリックします。
16. ルールの詳細を確認し、ルールの作成 を選択します。

デバイスヘルス通知を設定する (CLI)

Greengrass ステータス変更イベントが発生したときに Amazon SNS トピックを発行する EventBridge ルールを作成するには、次のステップを使用します。これにより、ウェブサーバー、E メールアドレス、その他のトピック受信者がイベントに応答できるようになります。

1. ルールを作成します。

    - デプロイステータス変更イベント用。

```
aws events put-rule \
  --name TestRule \
  --event-pattern "{\"source\": [\"aws.greengrass\"], \"detail-type\":
 [\"Greengrass V2 Effective Deployment Status Change\"]}"
```

- コンポーネントのステータス変更イベント用。

```
aws events put-rule \
  --name TestRule \
  --event-pattern "{\"source\": [\"aws.greengrass\"], \"detail-type\":
 [\"Greengrass V2 Installed Component Status Change\"]}"
```

パターンで省略されたプロパティは無視されます。

2. トピックをルールターゲットとして追加します。

- *topic-arn* を Amazon SNS トピックの ARN に置き換えます。

```
aws events put-targets \
  --rule TestRule \
  --targets "Id"="1","Arn"="topic-arn"
```

> **Note**
>
> Amazon EventBridge がターゲットトピックを呼び出せるようにするには、トピックにリソースベースのポリシーを追加する必要があります。詳細については、「Amazon ユーザーガイド」の「Amazon SNS アクセス許可 EventBridge 」を参照してください。

詳細については、「Amazon ユーザーガイド」の「 のイベントとイベントパターン EventBridge」を参照してください。 EventBridge

デバイスヘルス通知を設定する (AWS CloudFormation)

AWS CloudFormation テンプレートを使用して、Greengrass グループのデプロイの状態変更に関する通知を送信する EventBridge ルールを作成します。詳細については、「 ユーザーガイド」の「Amazon EventBridge リソースタイプのリファレンス」を参照してください。 AWS CloudFormation

以下も参照してください。

- デバイスへのデプロイのステータスを確認する
- 「Amazon ユーザーガイド」の「Amazon EventBridgeとは EventBridge 」

Greengrass コアデバイスのステータスを確認する

Greengrass コアデバイスは、ソフトウェアコンポーネントのステータスを AWS IoT Greengrass に報告します。各デバイスのヘルスサマリーと、各デバイス上にある各コンポーネントのステータスを確認することができます。

コアデバイスのヘルスステータスは次のとおりです。

- HEALTHY - AWS IoT Greengrass Core ソフトウェアとすべてのコンポーネントが、コアデバイス上で問題なく実行されています。
- UNHEALTHY - AWS IoT Greengrass Core ソフトウェアまたはコンポーネントが、コアデバイス上でエラー状態にあります。

> **Note**
>
> AWS IoT Greengrass は、ステータス更新を AWS クラウド に送信するタスクを各デバイスに任せています。デバイス上で AWS IoT Greengrass Core ソフトウェアが動作していない場合、またはデバイスが AWS クラウド に接続されていない場合、報告されたデバイスのステータスが現在のステータスを反映していない可能性があります。ステータスのタイムスタンプは、デバイスのステータスが最後に更新された日時を示しています。
> コアデバイスは、次のタイミングでステータス更新を送信します。
>
> - AWS IoT Greengrass Core ソフトウェアが開始したとき
> - コアデバイスが AWS クラウド からデプロイを受信したとき
> - コアデバイス上のコンポーネントのステータスが ERROREDまたは になった場合 BROKEN
> - ユーザーが設定した定期的な間隔 (デフォルトでは 24 時間)
>
> AWS IoT Greengrass Core v2.7.0 では、ローカルとクラウドの両方のデプロイが発生すると、コアデバイスがステータスの更新を送信します

トピック

コアデバイスのヘルス状態を確認する

個々のコアデバイスのステータスを確認できます。

コアデバイスのステータスを確認するには (AWS CLI)

- 次のコマンドを実行し、デバイスのステータスを取得します。*coreDeviceName* をクエリするコアデバイスの名前に置き換えます。

  ```
  aws greengrassv2 get-core-device --core-device-thing-name coreDeviceName
  ```

  レスポンスには、ステータスを含めたコアデバイスに関する情報が含まれています。

コアデバイスグループのヘルス状態を確認する

コアデバイスのグループ (モノグループ) のステータスを確認できます。

デバイスグループのステータスを確認するには (AWS CLI)

- 次のコマンドを実行し、複数のコアデバイスのステータスを取得します。コマンド内の ARN を、クエリするモノグループの ARN に置き換えます。

  ```
  aws greengrassv2 list-core-devices --thing-group-arn "arn:aws:iot:region:account-id:thinggroup/thingGroupName"
  ```

  レスポンスには、モノグループのコアデバイスのリストが含まれています。リストの各エントリには、コアデバイスのステータスが含まれます。

コアデバイスのコンポーネントステータスを確認する

コアデバイス上のソフトウェアコンポーネントのステータス (ライフサイクルステータスなど) を確認できます。コンポーネントのライフサイクルステータスの詳細については、「AWS IoT Greengrass コンポーネントを開発する」を参照してください。

コアデバイス上のコンポーネントのステータスを確認するには (AWS CLI)

- 次のコマンドを実行し、コアデバイス上のコンポーネントのステータスを取得します。*coreDeviceName* をクエリするコアデバイスの名前に置き換えます。

  ```
  aws greengrassv2 list-installed-components --core-device-thing-name coreDeviceName
  ```

  レスポンスには、コアデバイスで実行されるコンポーネントのリストが含まれています。リスト内の各エントリには、コンポーネントのライフサイクル状態が含まれています。これには、データのステータスがどの程度最新であるか、および Greengrass コアデバイスが特定のコンポーネントを含むメッセージを最後にクラウドに送信した日時が含まれます。レスポンスには、コンポーネントを Greengrass コアデバイスに導入した最新のデプロイソースも含まれています。

  > **Note**
  >
  > このコマンドは、Greengrass コアデバイスが実行するコンポーネントのページ分割されたリストを取得します。デフォルトでは、このリストには、他のコンポーネントの依存関係の中でデプロイされるコンポーネントは含まれません。`topologyFilter` パラメータを `ALL` に設定することで、応答に依存関係を含めることができます。

AWS Lambda 関数を実行する

> **Note**
>
> AWS IoT Greengrass は、現在 Windows コアデバイスにこの機能をサポートしていません。

AWS Lambda 関数を AWS IoT Greengrass コアデバイスで実行されるコンポーネントとしてインポートできます。このようにするのは、次のような場合です。

- コアデバイスにデプロイする Lambda 関数にアプリケーションコードがあります。
- AWS IoT Greengrass V2 コアデバイスで実行する AWS IoT Greengrass V1 アプリケーションがあります。詳細については、「ステップ 2: AWS IoT Greengrass V2 コンポーネントを作成してデプロイし、 AWS IoT Greengrass V1 アプリケーションを移行する」を参照してください。

Lambda 関数には以下のコンポーネントに依存関係が含まれています。関数のインポート時に、これらのコンポーネントを依存関係として定義する必要はありません。Lambda 関数コンポーネントをデプロイすると、デプロイにはこれらのLambdaコンポーネントの依存関係が含まれます。

- Lambda ランチャーコンポーネント (`aws.greengrass.LambdaLauncher`) は、プロセスと環境設定を処理します。
- -[Lambda manager component] (Lambda マネージャーコンポーネント) (`aws.greengrass.LambdaManager`) は、プロセス間通信とスケーリングを処理します。
- Lambda ランタイムコンポーネント (`aws.greengrass.LambdaRuntimes`) は、サポートされている各 Lambda ランタイムのアーティファクトを提供します。

トピック

要件

コアデバイスと Lambda 関数が AWS IoT Greengrass Core ソフトウェアで関数を実行するには、次の要件を満たす必要があります。

- コアデバイスは、Lambda 関数を実行するための要件を満たしている必要があります。コアデバイスが、コンテナ化された Lambda 関数を実行させる場合、そのデバイスは要件を満たす必要があります。詳細については、「Lambda 関数の要件」を参照してください。
- Lambda 関数が使用するプログラミング言語をコアデバイスにインストールする必要があります。

> **Tip**
>
> プログラミング言語をインストールするコンポーネントを作成し、そのコンポーネントを Lambda 関数コンポーネントの依存関係として指定できます。Greengrass は、Lambda でサポートされるすべてのバージョンの Python、Node.js、Java ランタイムをサポートします。Greengrass は、非推奨となった Lambda ランタイムバージョンに追加の制限を適用しません。これらの非推奨になったランタイムを使用する Lambda 関数は AWS IoT Greengrass で実行できますが、AWS Lambda で作成することはできません。Lambda ランタイムの AWS IoT Greengrass サポートの詳細については、「AWS Lambda 関数を実行する」を参照してください。

Lambda 関数のライフサイクルを設定する

Greengrass Lambda 関数ライフサイクルは、関数が開始する時期とどのようにコンテナを作成して使用するかを定義します。また、ライフサイクルでは、AWS IoT Greengrass Core ソフトウェアがファンクションハンドラーの外部にある変数や前処理ロジックをどのように保持するかも決定されます。

AWS IoT Greengrass は、オンデマンド (デフォルト) および長い存続期間のライフサイクルをサポートしています。

- オンデマンド 関数は、呼び出されたときに起動し、実行するタスクが残っていないときに停止します。関数を呼び出すたびに、サンドボックスとも呼ばれる個別のコンテナが作成され、既存のコンテナが再利用可能でない限り、呼び出しが処理されます。どのコンテナでも、関数に送信したデータを処理することがあります。

  オンデマンド関数の複数の呼び出しを同時に実行できます。

関数ハンドラーの外部で定義した変数や前処理ロジックは、新しいコンテナが作成されるときに保持されません。

- 長い存続期間 (あるいは固定された) 関数は、AWS IoT Greengrass Core ソフトウェアが起動して単一のコンテナで実行されたときに開始されます。同じコンテナが、関数に送信するすべてのデータを処理します。

  AWS IoT Greengrass Core ソフトウェアが以前の呼び出しを実行するまでキュー状態になります。

  関数ハンドラーの外部で定義する変数と事前処理ロジックは、このハンドラーの毎回の呼び出しのために保持されます。

  初期入力なしで作業を開始する必要がある場合は、長い存続期間の Lambda 関数を使用してください。例えば、存続期間が長い関数は、機械学習モデルを読み込んで処理を開始し、関数がデバイスデータを受信したときに準備が整うようにすることができます。

  > **Note**
  >
  > 長い存続期間の関数には、ハンドラーの各呼び出しに関連付けられたタイムアウトがあります。無期限に実行されるコードを呼び出す場合は、ハンドラーの外部で開始する必要があります。関数の初期化を妨げる可能性のあるブロッキングコードがハンドラーの外部にないことを確認してください。
  > これらの機能は、デプロイ中や 再起動中など、AWS IoT Greengrass Core ソフトウェアが停止しない限り実行されます。これらの関数は、関数がキャッチされない例外に遭遇した場合、そのメモリ制限を超過した場合、またはハンドラータイムアウトなどのエラー状態に入った場合は実行されません。

コンテナの再利用の詳細については、「AWS コンピューティングブログ」で「AWS Lambda のコンテナの再利用について」を参照してください。

Lambda 関数のコンテナ化を設定する

デフォルトでは、Lambda 関数は AWS IoT Greengrass コンテナ内で実行されます。Greengrass コンテナは、関数とホスト間の分離を提供します。この分離により、ホストとコンテナ内の関数の両方でセキュリティが向上します。

ユースケースでコンテナ化せずに実行する必要がある場合を除き、Lambda 関数を Greengrass コンテナで実行することをお勧めします。Greengrass コンテナで Lambda 関数を実行することで、リソースへのアクセスを制限する方法をより細かく制御できます。

以下の場合、コンテナ化を使用しないで Lambda 関数を実行できます。

- コンテナモードをサポートしていないデバイスで AWS IoT Greengrass を実行する場合。例えば、特殊な Linux ディストリビューションを使用する場合や 以前のカーネルバージョンが古くなった場合などです。
- 独自の OverlayFS がある別のコンテナ環境の Lambda 関数を、Greengrass コンテナで実行すると、OverlayFS の競合が発生する場合。
- デプロイ時に決定できない、またはデプロイ後にパスが変更される可能性のあるローカルリソースにアクセスする必要があります。このリソースの例としては、プラガブルデバイスがあります。
- プロセスとして作成された以前のアプリケーションがあり、それを Greengrass コンテナで実行すると問題が発生します。

コンテナ化の相違点

コンテナ化	メモ
Greengrass コンテナ	• Lambda 関数を Greengrass コンテナで実行する場合、すべての AWS IoT Greengrass 機能が使用可能です。 • Greengrass コンテナで実行する Lambda 関数は、他の Lambda 関数のデプロイ済みコードにアクセスできません (同じシステムグループを使用している場合でも)。つまり、Lambda 関数は相互に分離された状態で実行されます。 • AWS IoT Greengrass Core ソフトウェアはすべての子プロセスを Lambda 関数と同じコンテナで実行するため、子プロセスは Lambda 関数が停止すると停止します。
コンテナなし	• 以下の機能は、コンテナ化されていない Lambda 関数では利用できません。 • Lambda 関数のメモリ制限。

コンテナ化	メモ
	• ローカルデバイスおよびボリュームリソース。Lambda 関数リソースとしてではなく、コアデバイスのファイルパスを使用してこれらのリソースにアクセスする必要があります。 • コンテナ化されていない Lambda 関数が機械学習リソースにアクセスする場合、リソース所有者を指定し、Lambda 関数ではなくリソースにアクセス権限を設定する必要があります。 • コンテナ化されていない Lambda 関数は、同じシステムグループで実行される他の Lambda 関数のデプロイ済みコードに対して読み取り専用アクセス許可があります。

Lambda 関数をデプロイするときにコンテナ化を変更すると、関数が期待通りに動作しない場合があります。Lambda 関数が、新しいコンテナ化設定で利用できなくなったローカルリソースを使用する場合、デプロイは失敗します。

- Greengrass コンテナでの実行からコンテナ化を使用しない実行へと Lambda 関数を変更すると、関数のメモリ制限は破棄されます。アタッチ済みのローカルリソースを使用する代わりに、ファイルシステムに直接アクセスする必要があります。Lambda 関数をデプロイする前に、すべてのアタッチ済みリソースを削除する必要があります。
- コンテナ化を使用しない実行からコンテナでの実行へと Lambda 関数を変更すると、Lambda 関数はファイルシステムに直接アクセスできなくなります。関数ごとにメモリ制限を定義するか、デフォルトの 16 MB のメモリ制限を受け入れる必要があります。これらの設定は、デプロイ時に Lambda 関数ごとに設定できます。

Lambda 関数コンポーネントのコンテナ化設定を変更するには、コンポーネントをデプロイするときに `containerMode` 設定パラメータの値を次のいずれかのオプションに設定します。

- `NoContainer` - コンポーネントは、分離されたランタイム環境では実行されません。

- `GreengrassContainer` – コンポーネントは、AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。

コンポーネントをデプロイおよび設定する方法の詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」および「コンポーネント設定の更新」を参照してください。

Lambda 関数をコンポーネントとしてインポートする (コンソール)

AWS IoT Greengrass コンソールを使用して Lambda 関数コンポーネントを作成する場合、既存の AWS Lambda 関数をインポートしてから、Greengrass デバイス上で動作するコンポーネントを作成するように設定します。

開始する前に、Greengrass デバイスで Lambda 関数を実行するための要件を確認してください。

タスク

- ステップ 1: インポートする Lambda 関数を選択する
- ステップ 2: Lambda 関数パラメータを設定する
- ステップ 3: (オプション) Lambda 関数がサポートするプラットフォームを指定します。
- ステップ 4: (オプション) Lambda 関数のコンポーネントの依存関係を指定します。
- ステップ 5: (オプション) コンテナで Lambda 関数を実行する
- ステップ 6: Lambda 関数コンポーネントを作成する

ステップ 1: インポートする Lambda 関数を選択する

1. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Components] (コンポーネント) を選択します。
2. [Components] (コンポーネント) ページで、[Create component] (コンポーネントの作成) を選択します。
3. [Component information] (コンポーネント情報) の [Create component] (コンポーネントの作成) ページで、[Import Lambda function] (Lambda 関数のインポート) を選択します。
4. [Lambda function] (Lambda 関数) で、インポートする Lambda 関数を検索して選択します。

   AWS IoT Greengrass は Lambda 関数の名前を使用してコンポーネントを作成します。

5. [Lambda function version] (Lambda 関数のバージョン) で、インポートするバージョンを選択します。$LATEST などの Lambda エイリアスを選択することはできません。

   AWS IoT Greengrass は Lambda 関数のバージョンを有効なセマンティックバージョンに変換してコンポーネントを作成します。例えば、関数のバージョンが 3 である場合、コンポーネントのバージョンは 3.0.0 になります。

ステップ 2: Lambda 関数パラメータを設定する

[Lambda function configuration] (Lambda 関数の設定) の [Create component] (コンポーネントの作成) ページで、Lambda 関数の実行に使用する次のパラメータを設定します。

1. (オプション) Lambda 関数が作業メッセージのためにサブスクライブするイベントソースのリストを追加します。この関数をローカルのパブリッシュ/サブスクライブメッセージと AWS IoT Core MQTT メッセージにサブスクライブするように、イベントソースに指定できます。Lambda 関数は、イベントソースからメッセージを受信したときに呼び出されます。

   > **Note**
   >
   > この関数を他の Lambda 関数またはコンポーネントからのメッセージにサブスクライブするには、この Lambda 関数コンポーネントをデプロイするときにレガシーサブスクリプションルーターコンポーネントをデプロイしてください。レガシーサブスクリプションルーターコンポーネントをデプロイするときは、Lambda 関数が使用するサブスクリプションを指定します。

   [Event sources] (イベントソース) で以下を実行して、イベントソースを追加します。

   a. 追加したイベントソースごとに、次のオプションを指定します。

      - [Topic] (トピック) – メッセージをサブスクライブするためのトピック。
      - [Type] (タイプ) – イベントソースのタイプ。次のオプションから選択します。
        - ローカルの公開/サブスクライブ – ローカルの公開/サブスクライブメッセージに対するサブスクライブ。

          Greengrass nucleus v2.6.0 以降、または Lambda マネージャー v2.2.5 以降を使用する場合、[Topic] (トピック) でタイプを指定する際に、MQTT トピックのワイルドカード (+ および #) を使用できます。

- AWS IoT Core MQTT – AWS IoT Core MQTT メッセージに対するサブスクライブ。

  [Topic] (トピック) でタイプを指定する際に、MQTT ワイルドカード (+ および #) を使用できます。

b. 別のイベントソースを追加するには、[Add event source] (イベントソースの追加) をクリックし、前の手順を繰り返します。イベントソースを削除するには、削除するイベントソースの横にある [Remove] (削除) を選択します。

2. [Timeout (seconds)] (タイムアウト (秒)) には、ピン留めされていない Lambda 関数のタイムアウトまでの最大実行時間 (秒) を入力してください。デフォルト値は 3 秒です。

3. [Pinned] (ピン留め) では、Lambda 関数コンポーネントをピン留めするかどうかを選択します。デフォルトは [True] (真) です。

   - ピン留めされた (または存続期間の長い) Lambda 関数は、AWS IoT Greengrass が起動すると起動し、独自のコンテナで実行し続けます。
   - ピン留めされていない (またはオンデマンドの) Lambda 関数は、作業項目を受け取ったときのみ開始し、アイドル状態のまま指定された最大アイドル時間を過ぎると終了します。関数に複数の作業項目がある場合、AWS IoT Greengrass Core ソフトウェアは関数のインスタンスを複数作成します。

4. (オプション) [Additional parameters] (その他のパラメータ) で、以下の Lambda 関数パラメータを設定します。

   - [Status timeout (seconds)] (ステータスタイムアウト (秒)) – Lambda 関数のコンポーネントが Lambda マネージャーコンポーネントにステータスの更新を送信する間隔 (秒)。このパラメータは、ピン留めされた関数にのみ適用されます。デフォルト値は 60 秒です。
   - [Maximum queue size] (最大キューサイズ) – Lambda 関数コンポーネントのメッセージキューの最大サイズ。AWS IoT Greengrass Core ソフトウェアは、Lambda 関数を実行して各メッセージを消費できるまで、FIFO (先入れ先出し) キューにメッセージを保存します。デフォルトはメッセージ 1,000 件です。
   - [Maximum number of instances] (インスタンスの最大数) – ピン留めされていない Lambda 関数が同時に実行できるインスタンスの最大数。デフォルトは 100 件です。
   - [Maximum idle time (seconds)] (最大アイドルタイム (秒)) – AWS IoT Greengrass Core ソフトウェアによりプロセスを停止されるまでに、ピン留めされていない Lambda 関数がアイドル状態でいられる最大時間 (秒)。デフォルト値は 60 秒です。
   - [Encoding type] (エンコードタイプ) – Lambda 関数がサポートするペイロードのタイプ。次のオプションから選択します。

- JSON
- バイナリ

デフォルトは JSON です。

5. (オプション) 実行時に Lambda 関数に渡すコマンドライン引数のリストを指定します。

   a. [Additional parameters, Process arguments] (追加パラメータ、プロセス引数) で [Add argument] (引数を追加) を選択します。

   b. 追加する引数ごとに、関数に渡す引数を入力します。

   c. 引数を削除するには、削除する引数の横にある [Remove] (削除) を選択します。

6. (オプション) 実行時に Lambda 関数を利用できる環境変数を指定してください。環境変数を使用すると、関数コードを変更することなく設定を保存および更新できます。

   a. [Additional parameters, Environment variables] (追加パラメータ、環境変数) で、[Add environment variables] (環境変数を追加) を選択します。

   b. 追加する環境変数ごとに、次のオプションを指定します。

      - [Key] (キー) – 変数名。
      - [Value] (値) – この変数のデフォルト値。

   c. 環境変数を削除するには、削除する環境変数の横にある [Remove] (削除) を選択します。

ステップ 3: (オプション) Lambda 関数がサポートするプラットフォームを指定します。

すべてのコアデバイスには、オペレーティングシステムとアーキテクチャの属性があります。Lambda 関数コンポーネントをデプロイすると、AWS IoT Greengrass Core ソフトウェアは、指定したプラットフォーム値とコアデバイスのプラットフォーム属性を比較して、そのデバイスで Lambda 関数がサポートされているかどうかを判断します。

> **Note**
>
> コアデバイスに Greengrass nucleus コンポーネント をデプロイする際、カスタムのプラットフォーム属性を指定することもできます。詳細については、「Greengrass nucleus コンポーネント」の「プラットフォームの上書きパラメータ」を参照してください。

[Lambda function configuration, Additional parameters, Platforms] (Lambda 関数の設定、追加パラメータ、プラットフォーム) で以下の手順を実行して、この Lambda 関数がサポートするプラットフォームを指定します。

1. プラットフォームごとに以下のオプションを指定します。

   - [Operating system] (オペレーティングシステム) – プラットフォームのオペレーティングシステムの名前。現在、サポートされている値は `linux` のみです。
   - [Architecture] (アーキテクチャ) – プラットフォームのプロセッサアーキテクチャ。サポートされている値は以下のとおりです。
     - `amd64`
     - `arm`
     - `aarch64`
     - `x86`
2. 別のプラットフォームを追加するには、[Add platform] (プラットフォームを追加) を選択して、先の手順を繰り返します。サポートされているプラットフォームを削除するには、削除するプラットフォームの横にある [Remove] (削除) を選択します。

ステップ 4: (オプション) Lambda 関数のコンポーネントの依存関係を指定します。

コンポーネントの依存関係は、関数が使用する追加の AWS が提供するコンポーネントまたはカスタムコンポーネントを識別します。Lambda 関数コンポーネントをデプロイすると、関数を実行するためのこれらの依存関係がデプロイに含まれます。

> **Important**
>
> AWS IoT Greengrass V1 で実行するために作成した Lambda 関数をインポートする際は、シークレット、ローカルシャドウ、ストリームマネージャーなど、関数が使用する機能に対して個々のコンポーネントの依存関係を定義する必要があります。これらのコンポーネントをハード依存関係として定義し、依存関係の状態が変化すると Lambda 関数コンポーネントが再起動するようにしてください。詳細については、「V1 Lambda 関数をインポートする」を参照してください。

[Lambda function configuration, Additional parameters, Component dependencies] (Lambda 関数の設定、追加パラメータ、コンポーネントの依存関係) で以下の手順を実行し、Lambda 関数のコンポーネントの依存関係を指定します。

1. [Add dependency] (依存関係の追加) を選択します。
2. 追加したコンポーネントの依存関係ごとに、以下のオプションを指定します。

   - [Component name] (コンポーネント名) – コンポーネント名。たとえば **`aws.greengrass.StreamManager`** と入力してストリームマネージャーコンポーネントを含めます。
   - [Version requirement] (バージョン要件) – このコンポーネントの依存関係において互換性のあるバージョンを識別する npm スタイルのセマンティックバージョン制約。1 つのバージョンまたはバージョンの範囲を指定できます。たとえば **`^1.0.0`** と入力すると、この Lambda 関数がストリームマネージャーコンポーネントの最初のメジャーバージョンのすべてのバージョンに依存することを指定できます。セマンティックバージョン制約の詳細については、npm semver calculator を参照してください。
   - [Type] (タイプ) – 依存関係のタイプ。次のオプションから選択します。
     - [Hard] (強) – 依存関係が状態を変化させると、Lambda 関数コンポーネントが再起動します。デフォルトではこれが選択されています。
     - [Soft] (弱) – 依存関係が状態を変化させても、Lambda 関数コンポーネントは再起動しません。
3. コンポーネントの依存関係を削除するには、コンポーネントの依存関係の横にある[Remove] (削除) を選択します。

ステップ 5: (オプション) コンテナで Lambda 関数を実行する

デフォルトでは、Lambda 関数は AWS IoT Greengrass Core ソフトウェア内の分離されたランタイム環境で実行されます。また、Lambda 関数を分離されないプロセスとして (つまり[No container] (コンテナなし) モードで) 実行することもできます。

[Linux process configuration] (Linux プロセス設定) で、[Isolation mode] (分離モード) について、Lambda 関数のコンテナ化を以下のオプションから選択します。

- [Greengrass container] (Greengrass コンテナ) – The Lambda 関数はコンテナで実行されます。デフォルトではこれが選択されています。
- [No container] (コンテナなし) – Lambda 関数は分離されないプロセスとして実行されます。

コンテナで Lambda 関数を実行する場合は、以下の手順で Lambda 関数のプロセス設定を行います。

1. コンテナを利用できるメモリの量とシステムリソース (ボリュームやデバイスなど) を設定します。

   [Container parameters] (コンテナパラメータ) で以下を行います。

   a. [Memory size] (メモリサイズ) にはコンテナに割り当てるメモリサイズを入力します。メモリサイズは [MB] または [KB] で指定できます。

   b. [Read-only sys folder] (読み取り専用の Sys フォルダ) について、コンテナがデバイスの `/sys` フォルダから情報を読み取れるかどうか指定します。デフォルトは [False] (偽) です。

2. (オプション) コンテナ化された Lambda 関数がアクセスできるローカルボリュームを設定します。ボリュームを定義すると、AWS IoT Greengrass Core ソフトウェアはソースファイルをコンテナ内の送信先にマウントします。

   a. [Volumes] (ボリューム) で、[Add volume] (ボリュームを追加) を選択します。

   b. 追加したボリュームごとに、次のオプションを指定します。

      - [Physical volume] (物理ボリューム) –コアデバイスのソースフォルダへのパス。
      - [Logical volume] (論理ボリューム) – コンテナの保存先フォルダへのパス。
      - [Permission] (アクセス権限) – (オプション) コンテナからソースフォルダへのアクセス権限。次のオプションから選択します。
        - [Read-only] (読み取り専用) – Lambda 関数はソースフォルダへの読み取り専用アクセス権が与えられます。デフォルトではこれが選択されています。
        - [Read-write] (読み取り/書き取り) – Lambda 関数はソースフォルダへの読み取り/書き込みアクセス権が与えられます。
      - [Add group owner] (グループ所有者の追加) – (オプション) Lambda 関数コンポーネントを実行するシステムグループをソースフォルダの所有者として追加するかどうか。デフォルトは [False] (偽) です。

   c. ボリュームを削除するには、削除するボリュームの横にある [Remove] (削除) を選択します。

3. (オプション) コンテナ化された Lambda 関数がアクセスできるローカルシステムデバイスを設定します。

   a. [Devices] (デバイス) で、[Add device] (デバイスの追加) を選択します。

b. 追加したデバイスごとに、次のオプションを指定します。

- [Mount path] (マウントパス) – コアデバイスのシステムデバイスへのパス。
- [Permission] (アクセス権限) – (オプション) コンテナからシステムデバイスへのアクセス権限。次のオプションから選択します。
  - [Read-only] (読み取り専用) – Lambda 関数はシステムデバイスへの読み取り専用アクセス権が与えられます。デフォルトではこれが選択されています。
  - [Read-write] (読み取り/書き取り) – Lambda 関数はソースフォルダへの読み取り/書き込みアクセス権が与えられます。
- [Add group owner] (グループ所有者の追加) – (オプション) Lambda 関数コンポーネントを実行するシステムグループをシステムデバイスの所有者として追加するかどうか。デフォルトは [False] (偽) です。

ステップ 6: Lambda 関数コンポーネントを作成する

Lambda 関数コンポーネントを設定した後は、[Create] (作成) をクリックして新しいコンポーネントの作成を完了します。

コアデバイスで Lambda 関数を実行するには、新しいコンポーネントをコアデバイスにデプロイします。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

Lambda 関数をコンポーネント (AWS CLI) としてインポートする

CreateComponentVersion オペレーションを使用して、Lambda 関数からコンポーネントを作成します。このオペレーションを呼び出すときは、`lambdaFunction` を指定して Lambda 関数をインポートします。

タスク

- ステップ 1: Lambda 関数の設定を定義する
- ステップ 2: Lambda 関数コンポーネントを作成する

ステップ 1: Lambda 関数の設定を定義する

1. `lambda-function-component.json` という名前のファイルを作成して、次の JSON オブジェクトをファイルにコピーします。`lambdaArn` を、インポートする Lambda 関数のARNに置き換えます。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1"
  }
}
```

> **Important**
>
> インポートする関数のバージョンが含まれた ARN を指定する必要があります。$LATEST のようなバージョン工イリアスは使用できません。

2. (オプション) コンポーネントの名前 (componentName) を指定します。このパラメータを省略すると、AWS IoT Greengrass は Lambda 関数の名前を使用してコンポーネントを作成します。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda"
  }
}
```

3. (オプション) コンポーネントのバージョン (componentVersion) を指定します。このパラメータを省略すると、AWS IoT Greengrass は Lambda 関数のバージョンを有効なセマンティックバージョンに変換してコンポーネントを作成します。例えば、関数のバージョンが `3` である場合、コンポーネントのバージョンは `3.0.0` になります。

> **Note**
>
> アップロードする各コンポーネントバージョンは一意である必要があります。アップロード後は編集できないため、必ず正しいコンポーネントバージョンをアップロードしてください。
>
> AWS IoT Greengrass はコンポーネントのセマンティックバージョンを使用します。セマンティックバージョンは、major.minor.patch といった番号システムに準拠します。例

えば、バージョン 1.0.0 は、コンポーネントの最初のメジャーリリースを表しています。詳細については、「セマンティックバージョンの仕様」を参照してください。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0"
  }
}
```

4. (オプション) この Lambda 関数がサポートするプラットフォームを指定します。各プラットフォームには、プラットフォームを識別するための属性のマップが含まれます。すべてのコアデバイスには、オペレーティングシステム (os) とアーキテクチャ (architecture) の属性があります。AWS IoT Greengrass Core ソフトウェアは、他のプラットフォーム属性を追加する場合があります。コアデバイスに Greengrass nucleus コンポーネント をデプロイする際、カスタムのプラットフォーム属性を指定することもできます。以下の操作を実行します。

   a. lambda-function-component.json の Lambda 関数にプラットフォームのリスト (componentPlatforms) を追加します。

   ```
   {
     "lambdaFunction": {
       "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
       "componentName": "com.example.HelloWorldLambda",
       "componentVersion": "1.0.0",
       "componentPlatforms": [

       ]
     }
   }
   ```

   b. サポートされている各プラットフォームをリストに追加します。各プラットフォームには識別用のフレンドリー name と属性のマップがあります。以下の例は、この関数が Linux を実行する x86 デバイスをサポートすることを指定しています。

   ```
   {
     "name": "Linux x86",
     "attributes": {
   ```

```
    "os": "linux",
    "architecture": "x86"
  }
}
```

lambda-function-component.json には以下の例のようなドキュメントが含まれる場合があります。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ]
  }
}
```

5. (オプション) Lambda 関数のコンポーネントの依存関係を指定します。Lambda 関数コンポーネントをデプロイすると、関数を実行するためのこれらの依存関係がデプロイに含まれます。

> **Important**
>
> AWS IoT Greengrass V1 で実行するために作成した Lambda 関数をインポートする際は、シークレット、ローカルシャドウ、ストリームマネージャーなど、関数が使用する機能に対して個々のコンポーネントの依存関係を定義する必要があります。これらのコンポーネントをハード依存関係として定義し、依存関係の状態が変化すると Lambda 関数コンポーネントが再起動するようにしてください。詳細については、「V1 Lambda 関数をインポートする」を参照してください。

以下の操作を実行します。

a. lambda-function-component.json の Lambda 関数にコンポーネントの依存関係 (componentDependencies) のマップを追加します。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {

    }
  }
}
```

b. 各コンポーネントの依存関係をマップに追加します。コンポーネント名をキーとして指定し、以下のパラメータを使用してオブジェクトを指定します。

- versionRequirement – コンポーネントの依存関係において互換性のあるバージョンを識別する npm スタイルのセマンティックバージョン制約。1 つのバージョンまたはバージョンの範囲を指定できます。セマンティックバージョン制約の詳細については、npm semver calculator を参照してください。
- dependencyType – (オプション) 依存関係のタイプ。次から選択します。
  - SOFT - 依存関係が状態を変化させても、Lambda 関数コンポーネントは再起動しません。
  - HARD - 依存関係が状態を変化させると、Lambda 関数コンポーネントが再起動します。

  デフォルトは HARD です。

次の例では、この Lambda 関数がストリームマネージャーコンポーネントの最初のメジャーバージョンのすべてのバージョンに依存することを指定しています。Lambda 関数コンポーネントは、ストリームマネージャーが再起動または更新されたときに再起動します。

```
{
  "aws.greengrass.StreamManager": {
    "versionRequirement": "^1.0.0",
    "dependencyType": "HARD"
  }
}
```

lambda-function-component.json には以下の例のようなドキュメントが含まれる場合があります。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    }
  }
}
```

6. (オプション) 関数の実行に使用する Lambda 関数のパラメータを設定します。環境変数、メッセージイベントソース、タイムアウト、コンテナ設定などのオプションを設定できます。以下の操作を実行します。

a. Lambda パラメータオブジェクト (componentLambdaParameters) を lambda-function-component.json の Lambda 関数に追加します。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {

    }
  }
}
```

b. (オプション) Lambda 関数が作業メッセージのためにサブスクライブするイベントソースを指定します。この関数をローカルのパブリッシュ/サブスクライブメッセージと AWS IoT Core MQTT メッセージにサブスクライブするように、イベントソースに指定できます。Lambda 関数は、イベントソースからメッセージを受信したときに呼び出されます。

> **Note**
>
> この関数を他の Lambda 関数またはコンポーネントからのメッセージにサブスクライブするには、この Lambda 関数コンポーネントをデプロイするときにレガシーサブスクリプションルーターコンポーネントをデプロイしてください。レガシーサブ

スクリプションルーターコンポーネントをデプロイするときは、Lambda 関数が使用するサブスクリプションを指定します。

以下の操作を実行します。

i. Lambda 関数のパラメータにイベントソース (eventSources) のリストを追加します。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [

      ]
    }
  }
}
```

ii. 各イベントソースをリストに追加します。各イベントソースには以下のパラメータがあります。

- topic – メッセージをサブスクライブするためのトピック。
- type – イベントソースのタイプ。次のオプションから選択します。

- PUB_SUB - ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブします。

  Greengrass nucleus v2.6.0 以降と、Lambda マネージャー v2.2.5 以降を使用する場合、`topic` のタイプを指定する際に (その中で)、MQTT トピックのワイルドカード (+ および #) を使用できます。

- IOT_CORE – AWS IoT Core MQTT メッセージをサブスクライブ。

  `topic` のタイプを指定する際に、MQTT ワイルドカード (+ および #) を使用できます。

次の例は、`hello/world/+` トピックのフィルターに一致するトピックに関する AWS IoT Core MQTT にサブスクライブします。

```
{
  "topic": "hello/world/+",
  "type": "IOT_CORE"
}
```

`lambda-function-component.json` は、次の例のようになります。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-
id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
```

```
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
      ]
    }
  }
}
```

c. (オプション) Lambda 関数パラメータオブジェクトで、次のパラメータのいずれかを指定します。

- `environmentVariables` – 実行時に Lambda 関数を利用できる環境変数のマップ。
- `execArgs` – 実行時に Lambda 関数に渡す引数のリスト。
- `inputPayloadEncodingType` – Lambda 関数がサポートするペイロードのタイプ。次のオプションから選択します。
  - `json`
  - `binary`

  デフォルト: `json`
- `pinned` – Lambda 関数をピン留めするかどうかを示します。デフォルトは `true` です。
  - ピン留めされた (または存続期間の長い) Lambda 関数は、AWS IoT Greengrass が起動すると起動し、独自のコンテナで実行し続けます。
  - ピン留めされていない (またはオンデマンドの) Lambda 関数は、作業項目を受け取ったときのみ開始し、アイドル状態のまま指定された最大アイドル時間を過ぎると終了します。関数に複数の作業項目がある場合、AWS IoT Greengrass Core ソフトウェアは関数のインスタンスを複数作成します。

  `maxIdleTimeInSeconds` を使用して、関数の最大アイドル時間を設定します。
- `timeoutInSeconds` - Lambda 関数がタイムアウトするまでの最大実行時間 (秒)。デフォルト値は 3 秒です。
- `statusTimeoutInSeconds` – Lambda 関数のコンポーネントが Lambda マネージャーコンポーネントにステータスの更新を送信する間隔 (秒)。このパラメータは、ピン留めされた関数にのみ適用されます。デフォルト値は 60 秒です。

- `maxIdleTimeInSeconds` – AWS IoT Greengrass Core ソフトウェアによりプロセスを停止されるまでに、ピン留めされていない Lambda 関数がアイドル状態でいられる最大時間 (秒)。デフォルト値は 60 秒です。
- `maxInstancesCount` – ピン留めされていない Lambda 関数が同時に実行できるインスタンスの最大数。デフォルトは 100 件です。
- `maxQueueSize` – Lambda 関数コンポーネントのメッセージキューの最大サイズ。AWS IoT Greengrass Core ソフトウェアは、Lambda 関数を実行して各メッセージを消費できるようになるまで、FIFO (first-in-first-out) キューにメッセージを保存します。デフォルトはメッセージ 1,000 件です。

`lambda-function-component.json` には以下の例のようなドキュメントが含まれる場合があります。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
      ],
```

```
      "environmentVariables": {
        "LIMIT": "300"
      },
      "execArgs": [
        "-d"
      ],
      "inputPayloadEncodingType": "json",
      "pinned": true,
      "timeoutInSeconds": 120,
      "statusTimeoutInSeconds": 30,
      "maxIdleTimeInSeconds": 30,
      "maxInstancesCount": 50,
      "maxQueueSize": 500
    }
  }
}
```

d. (オプション) Lambda 関数のコンテナ設定を行います。デフォルトでは、Lambda 関数は AWS IoT Greengrass Core ソフトウェア内の分離されたランタイム環境で実行されます。また、Lambda 関数を分離されないプロセスとして実行することもできます。コンテナで Lambda 関数を実行する場合は、コンテナのメモリサイズと Lambda 関数で利用できるシステムリソースを設定します。以下の操作を実行します。

   i. Linux プロセスパラメータオブジェクト (`linuxProcessParams`) を、`lambda-function-component.json` の Lambda パラメータオブジェクトに追加します

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
```

```
          "dependencyType": "HARD"
        }
      },
      "componentLambdaParameters": {
        "eventSources": [
          {
            "topic": "hello/world/+",
            "type": "IOT_CORE"
          }
        ],
        "environmentVariables": {
          "LIMIT": "300"
        },
        "execArgs": [
          "-d"
        ],
        "inputPayloadEncodingType": "json",
        "pinned": true,
        "timeoutInSeconds": 120,
        "statusTimeoutInSeconds": 30,
        "maxIdleTimeInSeconds": 30,
        "maxInstancesCount": 50,
        "maxQueueSize": 500,
        "linuxProcessParams": {

        }
      }
    }
  }
```

ii. (オプション) Lambda 関数をコンテナで実行するかどうかを指定します。isolationMode パラメータをプロセスパラメータオブジェクトに追加し、次のオプションから選択します。

- GreengrassContainer – Lambda 関数はコンテナで実行されます。
- NoContainer – Lambda 関数は分離されないプロセスとして実行されます。

デフォルトは GreengrassContainer です。

iii. (オプション) コンテナで Lambda 関数を実行する場合、コンテナを利用できるメモリの量とシステムリソース (ボリュームやデバイスなど) を設定できます。以下の操作を実行します。

A. コンテナパラメータオブジェクト (containerParams) を、lambda-function-component.json の Linux プロセスパラメータオブジェクトに追加します

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-
id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
      ],
      "environmentVariables": {
        "LIMIT": "300"
      },
      "execArgs": [
        "-d"
      ],
      "inputPayloadEncodingType": "json",
      "pinned": true,
      "timeoutInSeconds": 120,
      "statusTimeoutInSeconds": 30,
      "maxIdleTimeInSeconds": 30,
```

```
      "maxInstancesCount": 50,
      "maxQueueSize": 500,
      "linuxProcessParams": {
        "containerParams": {

        }
      }
    }
  }
}
```

B. (オプション) memorySizeInKB パラメータを追加して、コンテナのメモリサイズを指定します。デフォルトは 16,384 KB (16 MB) です。

C. (オプション) mountROSysfsパラメータを追加して、コンテナがデバイスの /sys フォルダから情報を読み取れるかどうかを指定します。デフォルトは false です。

D. (オプション) コンテナ化された Lambda 関数がアクセスできるローカルボリュームを設定します。ボリュームを定義すると、AWS IoT Greengrass Core ソフトウェアはソースファイルをコンテナ内の送信先にマウントします。以下の操作を実行します。

   I. ボリュームのリスト (volumes) をコンテナパラメータに追加します。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-
id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
```

```
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
      ],
      "environmentVariables": {
        "LIMIT": "300"
      },
      "execArgs": [
        "-d"
      ],
      "inputPayloadEncodingType": "json",
      "pinned": true,
      "timeoutInSeconds": 120,
      "statusTimeoutInSeconds": 30,
      "maxIdleTimeInSeconds": 30,
      "maxInstancesCount": 50,
      "maxQueueSize": 500,
      "linuxProcessParams": {
        "containerParams": {
          "memorySizeInKB": 32768,
          "mountROSysfs": true,
          "volumes": [

          ]
        }
      }
    }
  }
}
```

II. 各ボリュームをリストに追加します。各ボリュームには以下のパラメータがあります。

- sourcePath – コアデバイスのソースフォルダへのパス。
- destinationPath – コンテナの保存先フォルダへのパス。
- permission - (オプション) コンテナからソースフォルダへのアクセス権限。次のオプションから選択します。

- `ro` – Lambda 関数はソースフォルダへの読み取り専用アクセス権が与えられます。
- `rw` – Lambda 関数はソースフォルダへの読み取り/書き込みアクセス権が与えられます。

デフォルトは `ro` です。

- `addGroupOwner` – (オプション) Lambda 関数コンポーネントを実行するシステムグループをソースフォルダの所有者として追加するかどうか。デフォルトは `false` です。

`lambda-function-component.json` には以下の例のようなドキュメントが含まれる場合があります。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-
id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
```

```
      ],
      "environmentVariables": {
        "LIMIT": "300"
      },
      "execArgs": [
        "-d"
      ],
      "inputPayloadEncodingType": "json",
      "pinned": true,
      "timeoutInSeconds": 120,
      "statusTimeoutInSeconds": 30,
      "maxIdleTimeInSeconds": 30,
      "maxInstancesCount": 50,
      "maxQueueSize": 500,
      "linuxProcessParams": {
        "containerParams": {
          "memorySizeInKB": 32768,
          "mountROSysfs": true,
          "volumes": [
            {
              "sourcePath": "/var/data/src",
              "destinationPath": "/var/data/dest",
              "permission": "rw",
              "addGroupOwner": true
            }
          ]
        }
      }
    }
  }
}
```

E. (オプション) コンテナ化された Lambda 関数がアクセスできるローカルシステムデバイスを設定します。以下の操作を実行します。

   I. システムデバイスのリスト (devices) をコンテナパラメータに追加します。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-
id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
```

```
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
      ],
      "environmentVariables": {
        "LIMIT": "300"
      },
      "execArgs": [
        "-d"
      ],
      "inputPayloadEncodingType": "json",
      "pinned": true,
      "timeoutInSeconds": 120,
      "statusTimeoutInSeconds": 30,
      "maxIdleTimeInSeconds": 30,
      "maxInstancesCount": 50,
      "maxQueueSize": 500,
      "linuxProcessParams": {
        "containerParams": {
          "memorySizeInKB": 32768,
          "mountROSysfs": true,
          "volumes": [
            {
              "sourcePath": "/var/data/src",
              "destinationPath": "/var/data/dest",
              "permission": "rw",
```

```
                "addGroupOwner": true
              }
            ],
            "devices": [

            ]
          }
        }
      }
    }
}
```

II. 各システムデバイスをリストに追加します。各システムデバイスには以下のパラメータがあります。

- path – コアデバイスのシステムデバイスへのパス。
- permission – (オプション) コンテナからシステムデバイスへのアクセス権限。次のオプションから選択します。
  - ro – Lambda 関数はシステムデバイスへの読み取り専用アクセス権が与えられます。
  - rw – Lambda 関数はシステムデバイスへの読み取り/書き込みアクセス権が与えられます。

  デフォルトは ro です。
- addGroupOwner – (オプション) Lambda 関数コンポーネントを実行するシステムグループをシステムデバイスの所有者として追加するかどうか。デフォルトは false です。

lambda-function-component.json には以下の例のようなドキュメントが含まれる場合があります。

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-
id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
```

```
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
      ],
      "environmentVariables": {
        "LIMIT": "300"
      },
      "execArgs": [
        "-d"
      ],
      "inputPayloadEncodingType": "json",
      "pinned": true,
      "timeoutInSeconds": 120,
      "statusTimeoutInSeconds": 30,
      "maxIdleTimeInSeconds": 30,
      "maxInstancesCount": 50,
      "maxQueueSize": 500,
      "linuxProcessParams": {
        "containerParams": {
          "memorySizeInKB": 32768,
          "mountROSysfs": true,
          "volumes": [
            {
              "sourcePath": "/var/data/src",
              "destinationPath": "/var/data/dest",
              "permission": "rw",
              "addGroupOwner": true
            }
          ],
```

```
          "devices": [
            {
              "path": "/dev/sda3",
              "permission": "rw",
              "addGroupOwner": true
            }
          ]
        }
      }
    }
  }
}
```

7. (オプション) コンポーネントのタグ (tags) を追加します。詳細については、「AWS IoT Greengrass Version 2 リソースのタグ付け」を参照してください。

ステップ 2: Lambda 関数コンポーネントを作成する

1. 以下のコマンドを実行して、lambda-function-component.json から Lambda 関数コンポーネントを作成します。

```
aws greengrassv2 create-component-version --cli-input-json file://lambda-function-component.json
```

リクエストが成功すると、レスポンスは次の例のようになります。

```
{
  "arn":
 "arn:aws:greengrass:region:123456789012:components:com.example.HelloWorldLambda:versions:1
  "componentName": "com.example.HelloWorldLambda",
  "componentVersion": "1.0.0",
  "creationTimestamp": "Mon Dec 15 20:56:34 UTC 2020",
  "status": {
    "componentState": "REQUESTED",
    "message": "NONE",
    "errors": {}
  }
}
```

次のステップでコンポーネントの状態をチェックするために、出力から arn をコピーします。

2. コンポーネントを作成すると、その状態は REQUESTED になります。その後、AWS IoT Greengrass がコンポーネントがデプロイ可能であるかどうかを検証します。次のコマンドを実行して、コンポーネントのステータスを照会し、コンポーネントがデプロイ可能であることを確認します。arn を、前のステップで書き留めた ARN に置き換えます。

```
aws greengrassv2 describe-component \
  --arn "arn:aws:greengrass:region:account-
id:components:com.example.HelloWorldLambda:versions:1.0.0"
```

コンポーネントが検証されると、レスポンスでコンポーネントの状態が DEPLOYABLE であることが示されます。

```
{
  "arn": "arn:aws:greengrass:region:account-
id:components:com.example.HelloWorldLambda:versions:1.0.0",
  "componentName": "com.example.HelloWorldLambda",
  "componentVersion": "1.0.0",
  "creationTimestamp": "2020-12-15T20:56:34.376000-08:00",
  "publisher": "AWS Lambda",
  "status": {
    "componentState": "DEPLOYABLE",
    "message": "NONE",
    "errors": {}
  },
  "platforms": [
    {
      "name": "Linux x86",
      "attributes": {
        "architecture": "x86",
        "os": "linux"
      }
    }
  ]
}
```

コンポーネントが DEPLOYABLE になった後は、Lambda 関数をコアデバイスにデプロイできます。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

AWS IoT Device SDK を使用して Greengrass nucleus、その他のコンポーネント、および AWS IoT Core と通信する

コアデバイスで実行されているコンポーネントは、AWS IoT Greengrass Core プロセス間通信 (IPC) ライブラリを AWS IoT Device SDK で使用して、AWS IoT Greengrass nucleus やその他の Greengrass コンポーネントと通信できます。IPC を使用するカスタムコンポーネントを開発して実行するには、AWS IoT Device SDK を使用して AWS IoT Greengrass Core IPC サービスに接続し、IPC オペレーションを実行する必要があります。

IPC インターフェイスは、以下の 2 種類のオペレーションをサポートします。

- リクエスト/レスポンス

  コンポーネントは IPC サービスにリクエストを送信し、リクエストの結果を含むレスポンスを受け取ります。
- サブスクリプション

  コンポーネントはサブスクリプション要求を IPC サービスに送信し、イベントメッセージのストリームをレスポンスとして期待します。コンポーネントは、イベントメッセージ、エラー、およびストリームクロージャを処理するサブスクリプションハンドラーを提供します。AWS IoT Device SDK には、各 IPC オペレーションの正しいレスポンスとイベントタイプを持つハンドラーインターフェイスが含まれます。詳細については、「IPC イベントストリームへのサブスクライブ」を参照してください。

トピック

- コンポーネント設定とやり取り
- シークレット値を取得する
- ローカルシャドウとやり取り
- ローカルデプロイおよびコンポーネントの管理
- クライアントデバイスを認証して承認する

IPC クライアントのバージョン

それ以降の Java および Python SDK のバージョンでは、IPC クライアント V2 と呼ばれる IPC クライアントの改良版を、AWS IoT Greengrass が提供します。IPC クライアント V2

- IPC オペレーションの使用に必要なコードの記述を削減し、IPC クライアント V1 で発生する一般的なエラーの回避に役立ちます。
- サブスクリプションハンドラーコールバックを別のスレッドで呼び出すため、IPC 関数呼び出しの追加呼び出しを含めたブロッキングコードを、サブスクリプションハンドラーコールバック内で実行できるようになりました。IPC クライアント V1 は同じスレッドを使用して、IPC サーバーとの通信とサブスクリプションハンドラーの呼び出しを行います。
- Lambda 式 (Java の場合) または関数 (Python の場合) を使用して、サブスクリプションオペレーションを呼び出せます。IPC クライアント V1 では、サブスクリプションハンドラクラスの定義が必要です。
- 同期バージョンと非同期バージョンの IPC オペレーションを提供します。IPC クライアント V1 では、非同期バージョンのオペレーションのみ提供します。

これらの機能改善を利用するには、IPC クライアント V2 の使用をお勧めします。ただし、このドキュメントやオンラインコンテンツに掲載されている例の多くは、IPC クライアント V1 の使用方法のみ紹介しています。次の例とチュートリアルから、IPC クライアント V2 を使用するサンプルのコンポーネントを確認できます。

- PublishToTopic 例
- SubscribeToTopic 例
- チュートリアル: コンポーネントの更新を延期する Greengrass コンポーネントを開発する
- チュートリアル: MQTT 経由でローカル IoT デバイスとやり取りする

現在 AWS IoT Device SDK for C++ v2 では、IPC クライアント V1のみをサポートしています。

プロセス間通信でサポートされている SDK

AWS IoT Greengrass Core IPC ライブラリは以下の AWS IoT Device SDK バージョンに含まれています。

SDK	最小バージョン	使用方法	
AWS IoT Device SDK for Java v2	v1.6.0	「AWS IoT Device SDK for Java v2 (IPC クライアント V2) を使用する」を参照してください。	
AWS IoT Device SDK for Python v2	v1.9.0	「AWS IoT Device SDK for Python v2 (IPC クライアント V2) を使用する」を参照してください。	
AWS IoT Device SDK for C++ v2	v1.17.0	「AWS IoT Device SDK for C++ v2 を使用」を参照してください。	
AWS IoT Device SDK JavaScript v2 用	v1.12.0	「AWS IoT Device SDK for JavaScript v2 (IPC クライアント V1) の使用」を参照してください。	

AWS IoT Greengrass Core IPC サービスに接続する

カスタムコンポーネントでプロセス間通信を使用するには、AWS IoT Greengrass Core ソフトウェアが稼働する IPC サーバーソケットへの接続を作成する必要があります。AWS IoT Device SDK を任意の言語でダウンロードして使用するには、次のタスクを実行してください。

AWS IoT Device SDK for Java v2 (IPC クライアント V2) を使用する

AWS IoT Device SDK for Java v2 (IPC クライアント V2) を使用するには

1. AWS IoT Device SDK for Java v2 (v1.6.0 以降) をダウンロードします。
2. 以下のいずれかを行って、コンポーネントでカスタムコードを実行します。
   - AWS IoT Device SDK JAR ファイルとしてコンポーネントを構築し、この JAR ファイルをコンポーネント recipe で実行します。
   - AWS IoT Device SDK JAR をコンポーネントアーティファクトとして定義し、コンポーネント recipe でアプリケーションを実行するときに、そのアーティファクトをクラスパスに追加します。
3. 以下のコードを使用して IPC クライアントを作成します。

```
try (GreengrassCoreIPCClientV2 ipcClient =
 GreengrassCoreIPCClientV2.builder().build()) {
    // Use client.
} catch (Exception e) {
    LOGGER.log(Level.SEVERE, "Exception occurred when using IPC.", e);
    System.exit(1);
}
```

AWS IoT Device SDK for Python v2 (IPC クライアント V2) を使用する

AWS IoT Device SDK for Python v2 (IPC クライアント V2) を使用するには

1. AWS IoT Device SDK for Python (v1.9.0 以降) をダウンロードします。
2. SDK のインストール手順を、コンポーネントの recipe のインストールライフサイクルに追加します。
3. AWS IoT Greengrass Core IPC サービスへの接続を作成します。以下のコードを使用して IPC クライアントを作成します。

```
from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2

try:
    ipc_client = GreengrassCoreIPCClientV2()
    # Use IPC client.
except Exception:
```

```
    print('Exception occurred when using IPC.', file=sys.stderr)
    traceback.print_exc()
    exit(1)
```

AWS IoT Device SDK for C++ v2 を使用

AWS IoT Device SDK v2 for C++ をビルドするには、デバイスに以下のツールが必要です。

- C++ 11 以降
- CMake 3.1 以降
- 以下のいずれかのコンパイラ:
  - GCC 4.8 以降
  - Clang 3.9 以降
  - MSVC 2015 以降

AWS IoT Device SDK for C++ v2 を使用するには

1. AWS IoT Device SDK for C++ v2 (v1.17.0 以降) をダウンロードします。
2. README のインストール手順に従って、AWS IoT Device SDK for C++ v2 をソースからビルドします。
3. C++ ビルドツールで、前の手順でビルドした Greengrass IPC ライブラリ (`AWS::GreengrassIpc-cpp`) をリンクします。以下の `CMakeLists.txt` の例は、Greengrass IPC ライブラリを CMake でビルドしたプロジェクトにリンクします。

```
cmake_minimum_required(VERSION 3.1)
project (greengrassv2_pubsub_subscriber)

file(GLOB MAIN_SRC
        "*.h"
        "*.cpp"
        )
add_executable(${PROJECT_NAME} ${MAIN_SRC})

set_target_properties(${PROJECT_NAME} PROPERTIES
        LINKER_LANGUAGE CXX
        CXX_STANDARD 11)
find_package(aws-crt-cpp PATHS ~/sdk-cpp-workspace/build)
find_package(EventstreamRpc-cpp PATHS ~/sdk-cpp-workspace/build)
```

```
find_package(GreengrassIpc-cpp PATHS ~/sdk-cpp-workspace/build)
target_link_libraries(${PROJECT_NAME} AWS::GreengrassIpc-cpp)
```

4. コンポーネントコードで AWS IoT Greengrass Core IPC サービスへの接続を作成して、IPC クライアント (`Aws::Greengrass::GreengrassCoreIpcClient`) を作成します。IPC の接続、切断、およびエラーイベントを処理する IPC 接続ライフサイクルハンドラーを定義する必要があります。次の例は、IPC クライアントの接続、切断、およびエラーの発生時に出力される IPC クライアントおよび IPC 接続ライフサイクルハンドラーを作成します。

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
    }

    void OnDisconnectCallback(RpcError error) override {
        std::cout << "OnDisconnectCallback: " << error.StatusToString() <<
 std::endl;
        exit(-1);
    }

    bool OnErrorCallback(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
};

int main() {
    // Create the IPC client.
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
```

```
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
 connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    // Use the IPC client to create an operation request.

    // Activate the operation request.
    auto activate = operation.Activate(request, nullptr);
    activate.wait();

    // Wait for Greengrass Core to respond to the request.
    auto responseFuture = operation.GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
 std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from
 Greengrass Core." << std::endl;
        exit(-1);
    }

    // Check the result of the request.
    auto response = responseFuture.get();
    if (response) {
        std::cout << "Successfully published to topic: " << topic << std::endl;
    } else {
        // An error occurred.
        std::cout << "Failed to publish to topic: " << topic << std::endl;
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            std::cout << "Operation error: " << error->GetMessage().value() <<
 std::endl;
        } else {
            std::cout << "RPC error: " << response.GetRpcError() << std::endl;
        }
        exit(-1);
    }

    return 0;
}
```

5. コンポーネントでカスタムコードを実行するには、コードをバイナリアーティファクトとしてビルドし、コンポーネント recipe でバイナリアーティファクトを実行します。OWNER へのアー

ティファクトの Execute 権限を設定して、AWS IoT Greengrass Core ソフトウェアがバイナリアーティファクトを実行できるようにします。

コンポーネント recipe の Manifests セクションは、次の例のようになります。

JSON

```
{
  ...
  "Manifests": [
    {
      "Lifecycle": {
        "run": "{artifacts:path}/greengrassv2_pubsub_subscriber"
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.PubSubSubscriberCpp/1.0.0/greengrassv2_pubsub_subscriber",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

YAML

```
...
Manifests:
  - Lifecycle:
      run: {artifacts:path}/greengrassv2_pubsub_subscriber
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.PubSubSubscriberCpp/1.0.0/greengrassv2_pubsub_subscriber
        Permission:
          Execute: OWNER
```

AWS IoT Device SDK for JavaScript v2 (IPC クライアント V1) の使用

NodeJS で使用する AWS IoT Device SDK JavaScriptv2 用の を構築するには、デバイスに次のツールが必要です。 NodeJS

- NodeJS 10.0 以降
  - node -v を実行して Node のバージョンを確認します。
- CMake 3.1 以降

AWS IoT Device SDK for JavaScript v2 (IPC クライアント V1) を使用するには

1. AWS IoT Device SDK for JavaScript v2 (v1.12.10 以降) をダウンロードします。
2. README のインストール手順に従って、ソースから AWS IoT Device SDK for JavaScript v2 を構築します。
3. AWS IoT Greengrass Core IPC サービスへの接続を作成します。次の手順を実行して、IPC クライアントを作成し、接続を確立します。
4. 以下のコードを使用して IPC クライアントを作成します。

```
import * as greengrascoreipc from 'aws-iot-device-sdk-v2';

let client = greengrascoreipc.createClient();
```

5. 次のコードを使用して、コンポーネントから Greengrass nucleus への接続を確立します。

```
await client.connect();
```

コンポーネントに IPC オペレーションの実行を許可する

カスタムコンポーネントが IPC オペレーションの一部を使用できるようにするには、コンポーネントが特定のリソースに対しオペレーションを実行できるようにする承認ポリシーを定義する必要があります。各承認ポリシーは、ポリシーが許可するオペレーションのリストとリソースのリストを定義します。たとえば、パブリッシュ/サブスクライブメッセージング IPC サービスは、トピックリソースの発行およびサブスクライブのオペレーションを定義します。* ワイルドカードを使用すると、すべてのオペレーションまたはすべてのリソースへのアクセスを許可できます。

承認ポリシーは accessControl 設定パラメータで定義しますが、これはコンポーネント recipe で、またはコンポーネントをデプロイするときに設定できます。accessControl オブジェクト

は、IPC サービス識別子を承認ポリシーのリストにマッピングします。各 IPC サービスに対しては、複数の承認ポリシーを定義してアクセスを制御できます。各承認ポリシーにはポリシー ID があり、すべてのコンポーネントで一意である必要があります。

> **Tip**
>
> 一意のポリシー ID を作成するには、コンポーネント名、IPC サービス名、およびカウンターを組み合わせることができます。たとえば `com.example.HelloWorld` という名前のコンポーネントには、以下の ID を持つ 2 つのパブリッシュ/サブスクライブ承認ポリシーを定義できます。
>
> - `com.example.HelloWorld:pubsub:1`
> - `com.example.HelloWorld:pubsub:2`

承認ポリシーは以下の形式を使用します。このオブジェクトは `accessControl` 設定パラメータです。

JSON

```
{
  "IPC service identifier": {
    "policyId": {
      "policyDescription": "description",
      "operations": [
        "operation1",
        "operation2"
      ],
      "resources": [
        "resource1",
        "resource2"
      ]
    }
  }
}
```

YAML

```
IPC service identifier:
  policyId:
    policyDescription: description
```

```
    operations:
      - operation1
      - operation2
    resources:
      - resource1
      - resource2
```

承認ポリシー内のワイルドカード

IPC 承認ポリシーの resources エレメントで * ワイルドカードを使うと、1 つの承認ポリシーにある複数のリソースにアクセスできます。

- Greengrass nucleus の全てのバージョンにおいて、リソースとして * の文字を指定すると、全てのリソースにアクセスできます。
- Greengrass nucleus の v2.6.0 以降では、リソースで * の文字を指定すると、どの文字列の組み合わせにも一致します。例えば factory/1/devices/Thermostat*/status と指定すると、工場内のサーモスタットデバイスのうち、名前が Thermostat で始まるすべてのデバイスのステータストピックへアクセスできます。

AWS IoT Core MQTT IPC サービスの承認ポリシーを定義する場合、MQTT ワイルドカード (+ および #) を使用すると、複数のリソースに一致します。詳細については、「AWS IoT Core MQTT IPC 承認ポリシーにおける MQTT ワイルドカード」を参照してください。

承認ポリシーの recipe 変数

Greengrass nucleus v2.6.0 以降を使用していて、Greengrass nucleus interpolateComponentConfigurationの設定オプションを に設定した場合はtrue、承認ポリシーで {iot:thingName} recipe 変数を使用できます。MQTT トピックやデバイスシャドウなど、コアデバイスの名前を含む承認ポリシーが必要な場合、この recipe 変数を使用すると、複数のコアデバイスからなるグループに対して、1 つの承認ポリシーを設定できます。例えば、シャドウ IPC オペレーションのために、コンポーネントに次のリソースへのアクセスを許可することができます。

```
$aws/things/{iot:thingName}/shadow/
```

承認ポリシーの特殊文字

承認ポリシーで * または ? をリテラル文字として指定するには、エスケープシーケンスを使う必要があります。次のエスケープシーケンスは、その文字が持つ特別な意味の代わりに、文字のリテラル

値を使うように AWS IoT Greengrass Core ソフトウェアに指示します。例えば、 * の文字は任意の文字の組み合わせに一致するワイルドカードです。

リテラル文字	エスケープシーケンス	メモ
*	${*}	
?	${?}	現在 AWS IoT Greengrass は、任意の 1 文字に一致する ? ワイルドカードをサポートしていません。
$	${$}	このエスケープシーケンスを使用すると、${ を含むリソースに一致します。例えば、${resourceName} という名前のリソースに一致させるには、${${}{resourceName} と指定する必要があります。もしくは、$aws で始まるトピックへアクセスできるように、リテラルの $ を使って $ を含むリソースに一致させます。

承認ポリシーの例

次の承認ポリシーの例を参照して、コンポーネントの承認ポリシー設定の参考にできます。

Example 承認ポリシーを使用したコンポーネント recipe の例

次のコンポーネント recipe の例には、承認ポリシーを定義する accessControl オブジェクトが含まれます。このポリシーは com.example.HelloWorld コンポーネントを承認して test/topic トピックに発行します。

JSON

```
{
```

```
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.HelloWorld",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.HelloWorld:pubsub:1": {
            "policyDescription": "Allows access to publish to test/topic.",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "test/topic"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "run": "java -jar {artifacts:path}/HelloWorld.jar"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: A component that publishes messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
```

```
        "com.example.HelloWorld:pubsub:1":
          policyDescription: Allows access to publish to test/topic.
          operations:
            - "aws.greengrass#PublishToTopic"
          resources:
            - "test/topic"
Manifests:
  - Lifecycle:
      run: |-
        java -jar {artifacts:path}/HelloWorld.jar
```

Example 承認ポリシーを使用したコンポーネント設定更新の例

次のデプロイの設定更新の例では、承認ポリシーを定義する accessControl オブジェクトでコンポーネントを設定することが指定されています。このポリシーは com.example.HelloWorld コンポーネントを承認して test/topic トピックに発行します。

Console

マージする設定

```
{
  "accessControl": {
    "aws.greengrass.ipc.pubsub": {
      "com.example.HelloWorld:pubsub:1": {
        "policyDescription": "Allows access to publish to test/topic.",
        "operations": [
          "aws.greengrass#PublishToTopic"
        ],
        "resources": [
          "test/topic"
        ]
      }
    }
  }
}
```

AWS CLI

次のコマンドは、コアデバイスにデプロイを作成します。

```
aws greengrassv2 create-deployment --cli-input-json file://hello-world-
deployment.json
```

hello-world-deployment.json ファイルには、次の JSON ドキュメントが含まれています。

```
{
  "targetArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
  "deploymentName": "Deployment for MyGreengrassCore",
  "components": {
    "com.example.HelloWorld": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "merge": "{\"accessControl\":{\"aws.greengrass.ipc.pubsub\":
{\"com.example.HelloWorld:pubsub:1\":{\"policyDescription\":\"Allows access to
 publish to test/topic.\",\"operations\":[\"aws.greengrass#PublishToTopic\"],
\"resources\":[\"test/topic\"]}}}}"
      }
    }
  }
}
```

Greengrass CLI

次の Greengrass CLI コマンドは、コアデバイスにローカルデプロイを作成します。

```
sudo greengrass-cli deployment create \
  --recipeDir recipes \
  --artifactDir artifacts \
  --merge "com.example.HelloWorld=1.0.0" \
  --update-config hello-world-configuration.json
```

hello-world-configuration.json ファイルには、次の JSON ドキュメントが含まれています。

```
{
  "com.example.HelloWorld": {
    "MERGE": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.HelloWorld:pubsub:1": {
            "policyDescription": "Allows access to publish to test/topic.",
```

```
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "test/topic"
            ]
          }
        }
      }
    }
  }
}
```

IPC イベントストリームへのサブスクライブ

IPC オペレーションを使用して、Greengrass コアデバイスのイベントのストリームにサブスクライブできます。サブスクライブオペレーションを使用するには、サブスクリプションハンドラーを定義して、IPC サービスへのリクエストを作成します。これで IPC クライアントは、コアデバイスがコンポーネントにイベントメッセージをストリーミングするたびに、サブスクリプションハンドラーの関数が実行するようになります。

サブスクリプションを閉じると、イベントメッセージの処理を停止できます。これを行うには、サブスクリプションを開くときに使用したサブスクリプションオペレーションオブジェクトで、`closeStream()` (Java)、`close()` (Python)、または `Close()` (C++) を呼び出します。

AWS IoT Greengrass Core IPC サービスは、以下のサブスクライブオペレーションをサポートします。

- SubscribeToTopic
- SubscribeToIoTCore
- SubscribeToComponentUpdates
- SubscribeToConfigurationUpdate
- SubscribeToValidateConfigurationUpdates

トピック

- サブスクリプションハンドラーの定義
- サブスクリプションハンドラーの例

サブスクリプションハンドラーの定義

サブスクリプションハンドラーを定義するには、イベントメッセージ、エラー、およびストリームクロージャを処理するコールバック関数を定義します。IPC クライアント V1 を使用する場合は、クラス内でこれらの関数を定義する必要があります。最近の Java または Python SDK で利用可能な IPC クライアント V2 を使用している場合、サブスクリプションハンドラークラスを作成しなくても、これらの関数を定義できます。

Java

IPC クライアント V1 を使用している場合は、一般的な `software.amazon.awssdk.eventstreamrpc.StreamResponseHandler<StreamEventType>` インターフェイスを実装する必要があります。*StreamEventType* は、サブスクリプションオペレーションのイベントメッセージのタイプです。次の関数を定義して、イベントメッセージ、エラー、およびストリームクロージャを処理します。

IPC クライアント V2 を使用している場合は、サブスクリプションハンドラクラスの外部でこれらの関数を定義するか、Lambda 式を使います。

`void onStreamEvent(StreamEventType event)`

IPC クライアントが MQTT メッセージやコンポーネント更新通知などのイベントメッセージを受信したときに呼び出すコールバック。

`boolean onStreamError(Throwable error)`

ストリームエラーが発生したときに IPC クライアントが呼び出すコールバック。

エラーの結果としてサブスクリプションストリームを閉じるには true を返し、ストリームを開いたままにするには false を返します。

`void onStreamClosed()`

ストリームが閉じたときに IPC クライアントが呼び出すコールバック。

Python

IPC クライアント V1 を使用している場合、サブスクリプションオペレーションに対応するストリームレスポンスハンドラークラスを拡張する必要があります。AWS IoT Device SDK には、サブスクリプションオペレーションごとにサブスクリプションハンドラークラスが含まれています。*StreamEventType* は、サブスクリプションオペレーションのイベントメッセージのタイプ

です。次の関数を定義して、イベントメッセージ、エラー、およびストリームクロージャを処理します。

IPC クライアント V2 を使用している場合は、サブスクリプションハンドラクラスの外部でこれらの関数を定義するか、Lambda 式を使います。

```
def on_stream_event(self, event: StreamEventType) -> None
```

IPC クライアントが MQTT メッセージやコンポーネント更新通知などのイベントメッセージを受信したときに呼び出すコールバック。

```
def on_stream_error(self, error: Exception) -> bool
```

ストリームエラーが発生したときに IPC クライアントが呼び出すコールバック。

エラーの結果としてサブスクリプションストリームを閉じるには true を返し、ストリームを開いたままにするには false を返します。

```
def on_stream_closed(self) -> None
```

ストリームが閉じたときに IPC クライアントが呼び出すコールバック。

C++

サブスクリプションオペレーションに対応するストリームレスポンスハンドラークラスから派生したクラスを実装します。AWS IoT Device SDK には、サブスクリプションオペレーションごとにサブスクリプションハンドラーベースクラスが含まれています。*StreamEventType* は、サブスクリプションオペレーションのイベントメッセージのタイプです。次の関数を定義して、イベントメッセージ、エラー、およびストリームクロージャを処理します。

```
void OnStreamEvent(StreamEventType *event)
```

IPC クライアントが MQTT メッセージやコンポーネント更新通知などのイベントメッセージを受信したときに呼び出すコールバック。

```
bool OnStreamError(OperationError *error)
```

ストリームエラーが発生したときに IPC クライアントが呼び出すコールバック。

エラーの結果としてサブスクリプションストリームを閉じるには true を返し、ストリームを開いたままにするには false を返します。

```
void OnStreamClosed()
```

ストリームが閉じたときに IPC クライアントが呼び出すコールバック。

JavaScript

サブスクリプションオペレーションに対応するストリームレスポンスハンドラークラスから派生したクラスを実装します。AWS IoT Device SDK には、サブスクリプションオペレーションごとにサブスクリプションハンドラーベースクラスが含まれています。*StreamEventType* は、サブスクリプションオペレーションのイベントメッセージのタイプです。次の関数を定義して、イベントメッセージ、エラー、およびストリームクロージャを処理します。

`on(event: 'ended', listener: StreamingOperationEndedListener)`

ストリームが閉じたときに IPC クライアントが呼び出すコールバック。

`on(event: 'streamError', listener: StreamingRpcErrorListener)`

ストリームエラーが発生したときに IPC クライアントが呼び出すコールバック。

エラーの結果としてサブスクリプションストリームを閉じるには true を返し、ストリームを開いたままにするには false を返します。

`on(event: 'message', listener: (message: InboundMessageType) => void)`

IPC クライアントが MQTT メッセージやコンポーネント更新通知などのイベントメッセージを受信したときに呼び出すコールバック。

サブスクリプションハンドラーの例

次の例は、SubscribeToTopic オペレーションと、ローカルのパブリッシュ/サブスクライブメッセージにサブスクライブするためのサブスクリプションハンドラーの使用方法を示します。

Java (IPC client V2)

Example 例: ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブする

```
package com.aws.greengrass.docs.samples.ipc;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientV2;
import software.amazon.awssdk.aws.greengrass.SubscribeToTopicResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.*;

import java.nio.charset.StandardCharsets;
import java.util.Optional;

public class SubscribeToTopicV2 {
```

```
   public static void main(String[] args) {
       String topic = args[0];
       try (GreengrassCoreIPCClientV2 ipcClient =
GreengrassCoreIPCClientV2.builder().build()) {
           SubscribeToTopicRequest request = new
SubscribeToTopicRequest().withTopic(topic);
           GreengrassCoreIPCClientV2.StreamingResponse<SubscribeToTopicResponse,
                   SubscribeToTopicResponseHandler> response =
                   ipcClient.subscribeToTopic(request,
SubscribeToTopicV2::onStreamEvent,
                           Optional.of(SubscribeToTopicV2::onStreamError),
                           Optional.of(SubscribeToTopicV2::onStreamClosed));
           SubscribeToTopicResponseHandler responseHandler =
response.getHandler();
           System.out.println("Successfully subscribed to topic: " + topic);

           // Keep the main thread alive, or the process will exit.
           try {
               while (true) {
                   Thread.sleep(10000);
               }
           } catch (InterruptedException e) {
               System.out.println("Subscribe interrupted.");
           }

           // To stop subscribing, close the stream.
           responseHandler.closeStream();
       } catch (Exception e) {
           if (e.getCause() instanceof UnauthorizedError) {
               System.err.println("Unauthorized error while publishing to topic: "
+ topic);
           } else {
               System.err.println("Exception occurred when using IPC.");
           }
           e.printStackTrace();
           System.exit(1);
       }
   }

   public static void onStreamEvent(SubscriptionResponseMessage
subscriptionResponseMessage) {
       try {
```

```
            BinaryMessage binaryMessage =
 subscriptionResponseMessage.getBinaryMessage();
            String message = new String(binaryMessage.getMessage(),
 StandardCharsets.UTF_8);
            String topic = binaryMessage.getContext().getTopic();
            System.out.printf("Received new message on topic %s: %s%n", topic,
 message);
        } catch (Exception e) {
            System.err.println("Exception occurred while processing subscription
 response " +
                    "message.");
            e.printStackTrace();
        }
    }

    public static boolean onStreamError(Throwable error) {
        System.err.println("Received a stream error.");
        error.printStackTrace();
        return false; // Return true to close stream, false to keep stream open.
    }

    public static void onStreamClosed() {
        System.out.println("Subscribe to topic stream closed.");
    }
}
```

Python (IPC client V2)

Example 例: ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブする

```
import sys
import time
import traceback

from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2
from awsiot.greengrasscoreipc.model import (
    SubscriptionResponseMessage,
    UnauthorizedError
)


def main():
    args = sys.argv[1:]
    topic = args[0]
```

```
    try:
        ipc_client = GreengrassCoreIPCClientV2()
        # Subscription operations return a tuple with the response and the
 operation.
        _, operation = ipc_client.subscribe_to_topic(topic=topic,
 on_stream_event=on_stream_event,

 on_stream_error=on_stream_error, on_stream_closed=on_stream_closed)
        print('Successfully subscribed to topic: ' + topic)

        # Keep the main thread alive, or the process will exit.
        try:
            while True:
                time.sleep(10)
        except InterruptedError:
            print('Subscribe interrupted.')

        # To stop subscribing, close the stream.
        operation.close()
    except UnauthorizedError:
        print('Unauthorized error while subscribing to topic: ' +
              topic, file=sys.stderr)
        traceback.print_exc()
        exit(1)
    except Exception:
        print('Exception occurred', file=sys.stderr)
        traceback.print_exc()
        exit(1)


def on_stream_event(event: SubscriptionResponseMessage) -> None:
    try:
        message = str(event.binary_message.message, 'utf-8')
        topic = event.binary_message.context.topic
        print('Received new message on topic %s: %s' % (topic, message))
    except:
        traceback.print_exc()


def on_stream_error(error: Exception) -> bool:
    print('Received a stream error.', file=sys.stderr)
    traceback.print_exc()
    return False  # Return True to close stream, False to keep stream open.
```

```
def on_stream_closed() -> None:
    print('Subscribe to topic stream closed.')


if __name__ == '__main__':
    main()
```

C++

Example 例: ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブする

```
#include <iostream>

#include </crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class SubscribeResponseHandler : public SubscribeToTopicStreamHandler {
    public:
        virtual ~SubscribeResponseHandler() {}

    private:
        void OnStreamEvent(SubscriptionResponseMessage *response) override {
            auto jsonMessage = response->GetJsonMessage();
            if (jsonMessage.has_value() &&
 jsonMessage.value().GetMessage().has_value()) {
                auto messageString =
 jsonMessage.value().GetMessage().value().View().WriteReadable();
                // Handle JSON message.
            } else {
                auto binaryMessage = response->GetBinaryMessage();
                if (binaryMessage.has_value() &&
 binaryMessage.value().GetMessage().has_value()) {
                    auto messageBytes = binaryMessage.value().GetMessage().value();
                    std::string messageString(messageBytes.begin(),
 messageBytes.end());
                    // Handle binary message.
                }
            }
        }
```

```
        bool OnStreamError(OperationError *error) override {
            // Handle error.
            return false; // Return true to close stream, false to keep stream open.
        }

        void OnStreamClosed() override {
            // Handle close.
        }
};

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }

    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }

    bool OnErrorCallback(RpcError error) override {
        // Handle IPC service connection error.
        return true;
    }
};

int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
 connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    String topic("my/topic");
    int timeout = 10;

    SubscribeToTopicRequest request;
```

```
    request.SetTopic(topic);

    //SubscribeResponseHandler streamHandler;
    auto streamHandler = MakeShared<SubscribeResponseHandler>(DefaultAllocator());
    auto operation = ipcClient.NewSubscribeToTopic(streamHandler);
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
 std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
 Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (!response) {
        // Handle error.
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            (void)error;
            // Handle operation error.
        } else {
            // Handle RPC error.
        }
        exit(-1);
    }

    // Keep the main thread alive, or the process will exit.
    while (true) {
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    operation->Close();
    return 0;
}
```

JavaScript

Example 例: ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブする

```
import * as greengrasscoreipc from "aws-iot-device-sdk-v2/dist/greengrasscoreipc";
import {SubscribeToTopicRequest, SubscriptionResponseMessage} from "aws-iot-device-
sdk-v2/dist/greengrasscoreipc/model";
import {RpcError} from "aws-iot-device-sdk-v2/dist/eventstream_rpc";

class SubscribeToTopic {
    private ipcClient : greengrasscoreipc.Client
    private readonly topic : string;

    constructor() {
        // define your own constructor, e.g.
        this.topic = "<define_your_topic>";
        this.subscribeToTopic().then(r => console.log("Started workflow"));
    }

    private async subscribeToTopic() {
        try {
            this.ipcClient = await getIpcClient();

            const subscribeToTopicRequest : SubscribeToTopicRequest = {
                topic: this.topic,
            }

            const streamingOperation =
 this.ipcClient.subscribeToTopic(subscribeToTopicRequest, undefined); //
 conditionally apply options

            streamingOperation.on("message", (message: SubscriptionResponseMessage)
 => {
                // parse the message depending on your use cases, e.g.
                if(message.binaryMessage && message.binaryMessage.message) {
                    const receivedMessage =
 message.binaryMessage?.message.toString();
                }
            });

            streamingOperation.on("streamError", (error : RpcError) => {
                // define your own error handling logic
            })

            streamingOperation.on("ended", () => {
                // define your own logic
            })
```

```
            await streamingOperation.activate();

            // Keep the main thread alive, or the process will exit.
            await new Promise((resolve) => setTimeout(resolve, 10000))
        } catch (e) {
            // parse the error depending on your use cases
            throw e
        }
    }
}

export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}

// starting point
const subscribeToTopic = new SubscribeToTopic();
```

IPC ベストプラクティス

カスタムコンポーネントで IPC を使用する場合のベストプラクティスは、IPC クライアント V1 と IPC クライアント V2 とで異なります。使用する IPC クライアントのバージョンのベストプラクティスに従ってください。

IPC client V2

IPC クライアント V2 は、別のスレッドでコールバック関数を実行するため、IPC クライアント V1 と比較すると、IPC を使用する場合や、サブスクリプションハンドラ関数を記述する場合に、従うべきガイドラインが少なくなります。

- 1 つの IPC クライアントを再利用する

  IPC クライアントを作成したら、そのクライアントを開いたままにして、全ての IPC オペレーションに再利用します。複数のクライアントを作成すると、リソースが余分に消費され、リソースリークが発生する可能性があります。

- 例外を処理する

  IPC クライアント V2 は、サブスクリプションハンドラー関数でキャッチされない例外をログに記録します。コードで発生したエラーを処理するには、ハンドラー関数で例外をキャッチする必要があります。

IPC client V1

IPC クライアント V1 は単一のスレッドを使用し、IPC サーバーとの通信とサブスクリプションハンドラーの呼び出しを行います。サブスクリプションハンドラー関数を記述するときは、この同期動作を考慮する必要があります。

- 1 つの IPC クライアントを再利用する

  IPC クライアントを作成したら、そのクライアントを開いたままにして、全ての IPC オペレーションに再利用します。複数のクライアントを作成すると、リソースが余分に消費され、リソースリークが発生する可能性があります。

- ブロッキングコードを非同期で実行する

  IPC クライアント V1 は、スレッドがブロックされている間は、新しいリクエストの送信や新しいイベントメッセージの処理を行うことはできません。ブロッキングコードは、ハンドラー関数から実行する別個のスレッドで実行するべきです。ブロックコードには `sleep` 呼び出し、連続して実行されるループ、および完了までに時間がかかる同期 I/O リクエストなどがあります。

- 新しい IPC 要求を非同期で送信する

  IPC クライアント V1 はサブスクリプションハンドラー関数内から新しいリクエストを送信できません。これは、応答を待機するとリクエストによってハンドラー関数がブロックされるためです。IPC リクエストの送信は、ハンドラー関数から実行する別個のスレッドで行うべきです。

- 例外を処理する

IPC クライアント V1 は、キャッチされない例外はサブスクリプションハンドラー関数で処理しません。ハンドラー関数が例外をスローすると、サブスクリプションが閉じて、例外はコンポーネントログに表示されません。ハンドラー関数で例外をキャッチして、サブスクリプションを開いたままにし、コードで発生したエラーをログに記録してください。

ローカルメッセージをパブリッシュ/サブスクライブする

パブリッシュ/サブスクライブ (pubsub) メッセージを使用すると、トピックにメッセージを送受信できます。コンポーネントはメッセージをトピックに発行して、他のコンポーネントにメッセージを送信できます。その後、そのトピックをサブスクライブしているコンポーネントは、受信したメッセージに対応できます。

> **Note**
>
> このパブリッシュ/サブスクライブ IPC サービスを使用して、AWS IoT Core MQTT を公開またはサブスクライブすることはできません。AWS IoT Core MQTT とメッセージを交換する方法の詳細については、「AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする」を参照してください。

トピック

最小 SDK バージョン

以下の表に、ローカルトピックとメッセージの発行およびサブスクライブのやりとりを行う際に使用が必要となる AWS IoT Device SDK の最小バージョンを示します。

SDK	最小バージョン	
AWS IoT Device SDK for Java v2	v1.2.10	
AWS IoT Device SDK for Python v2	v1.5.3	
AWS IoT Device SDK for C++ v2	v1.17.0	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

認証

ローカルのパブリッシュ/サブスクライブメッセージングをカスタムコンポーネントで使用するには、コンポーネントがトピックにメッセージを送受信できるようにする承認ポリシーを定義する必要があります。承認ポリシーの定義については、「コンポーネントに IPC オペレーションの実行を許可する」を参照してください。

パブリッシュ/サブスクライブメッセージングの承認ポリシーには以下のプロパティがあります。

IPC サービス識別子: `aws.greengrass.ipc.pubsub`

操作	説明	リソース
`aws.greengrass#PublishToTopic`	コンポーネントが指定したトピックにメッセージを発行できるようにします。	`test/topic` などのトピック文字列。トピック内の任意の文字の組み合わせに一致させるには * を使用します。 このトピック文字列は、MQTT トピックのワイルドカード (# および +) をサポートしません。

操作	説明	リソース
aws.greengrass#SubscribeToTopic	コンポーネントが、指定したトピックに関するメッセージをサブスクライブできるようにします。	test/topic などのトピック文字列。トピック内の任意の文字の組み合わせに一致させるには * を使用します。 Greengrass nucleus v2.6.0 以降では、MQTT トピックワイルドカード (# および +) を含むトピックをサブスクライブできます。このトピック文字列は MQTT トピックのワイルドカードを文字そのものとしてサポートします。例えば、コンポーネントの承認ポリシーで test/topic/# へのアクセス権が付与されている場合、コンポーネントは test/topic/# をサブスクライブできますが、test/topic/filter はサブスクライブできません。

操作	説明	リソース
*	コンポーネントが、指定したトピックのメッセージを発行およびサブスクライブできるようにします。	`test/topic` などのトピック文字列。トピック内の任意の文字の組み合わせに一致させるには * を使用します。 Greengrass nucleus v2.6.0 以降では、MQTT トピックワイルドカード (# および +) を含むトピックをサブスクライブできます。このトピック文字列は MQTT トピックのワイルドカードを文字そのものとしてサポートします。例えば、コンポーネントの承認ポリシーで `test/topic/#` へのアクセス権が付与されている場合、コンポーネントは `test/topic/#` をサブスクライブできますが、`test/topic/filter` はサブスクライブできません。

承認ポリシーの例

次の承認ポリシーの例を参照して、コンポーネントの承認ポリシーの設定に役立てることができます。

Example 承認ポリシーの例

以下の承認ポリシーの例では、コンポーネントがすべてのトピックを公開およびサブスクライブすることを許可します。

```
{
  "accessControl": {
    "aws.greengrass.ipc.pubsub": {
      "com.example.MyLocalPubSubComponent:pubsub:1": {
```

```
        "policyDescription": "Allows access to publish/subscribe to all topics.",
        "operations": [
          "aws.greengrass#PublishToTopic",
          "aws.greengrass#SubscribeToTopic"
        ],
        "resources": [
          "*"
        ]
      }
    }
  }
}
```

PublishToTopic

トピックへのメッセージの発行

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`topic`

メッセージの発行先として指定するトピック。

`publishMessage` (Python: `publish_message`)

発行するメッセージ。このオブジェクト (`PublishMessage`) には、次の情報が含まれます。`jsonMessage` と `binaryMessage` のどちらか 1 つを指定する必要があります。

`jsonMessage` (Python: `json_message`)

(オプション) JSON メッセージ。このオブジェクト (`JsonMessage`) には、次の情報が含まれます。

`message`

オブジェクトとしての JSON メッセージ。

`context`

メッセージが公開されたトピックなど、メッセージのコンテキスト。

この機能は、Greengrass nucleus コンポーネントの v2.6.0 以降で利用できます。次の表に、メッセージコンテキストへのアクセスに使用する必要がある AWS IoT Device SDK の最小バージョンを示します。

SDK	最小バージョン	
AWS IoT Device SDK for Java v2	v1.9.3	
AWS IoT Device SDK for Python v2	v1.11.3	
AWS IoT Device SDK for C++ v2	v1.18.4	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアは PublishToTopic および SubscribeToTopic オペレーションで同じメッセージオブジェクトを使用します。AWS IoT Greengrass Core ソフトウェアは、サブスクライブ時にこのコンテキストオブジェクトをメッセージに設定し、発行するメッセージ内のこのコンテキストオブジェクトを無視します。

このオブジェクト (MessageContext) には、次の情報が含まれます。

`topic`

メッセージが発行されたトピック。

`binaryMessage` (Python: `binary_message`)

(オプション) バイナリメッセージ。このオブジェクト (BinaryMessage) には、次の情報が含まれます。

`message`

BLOB としてのバイナリメッセージ。

`context`

メッセージが公開されたトピックなど、メッセージのコンテキスト。

この機能は、Greengrass nucleus コンポーネントの v2.6.0 以降で利用できます。次の表に、メッセージコンテキストへのアクセスに使用する必要がある AWS IoT Device SDK の最小バージョンを示します。

SDK	最小バージョン
AWS IoT Device SDK for Java v2	v1.9.3
AWS IoT Device SDK for Python v2	v1.11.3
AWS IoT Device SDK for C++ v2	v1.18.4
AWS IoT Device SDK for JavaScript v2	v1.12.0

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアは `PublishToTopic` および `SubscribeToTopic` オペレーションで同じメッセージオブジェクトを使用します。AWS IoT Greengrass Core ソフトウェアは、サブスクライブ時にこのコンテキストオブジェクトをメッセージに設定し、発行するメッセージ内のこのコンテキストオブジェクトを無視します。

このオブジェクト (`MessageContext`) には、次の情報が含まれます。

`topic`

メッセージが発行されたトピック。

レスポンス

このオペレーションはレスポンスで一切の情報を提供しません。

例

以下の例では、カスタムコンポーネントコードでこのオペレーションを呼び出す方法を示します。

Java (IPC client V2)

Example 例: バイナリメッセージを発行する

```
package com.aws.greengrass.docs.samples.ipc;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientV2;
import software.amazon.awssdk.aws.greengrass.model.BinaryMessage;
import software.amazon.awssdk.aws.greengrass.model.PublishMessage;
import software.amazon.awssdk.aws.greengrass.model.PublishToTopicRequest;
import software.amazon.awssdk.aws.greengrass.model.PublishToTopicResponse;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;

import java.nio.charset.StandardCharsets;

public class PublishToTopicV2 {

    public static void main(String[] args) {
        String topic = args[0];
        String message = args[1];
        try (GreengrassCoreIPCClientV2 ipcClient =
 GreengrassCoreIPCClientV2.builder().build()) {
            PublishToTopicV2.publishBinaryMessageToTopic(ipcClient, topic,
 message);
            System.out.println("Successfully published to topic: " + topic);
        } catch (Exception e) {
            if (e.getCause() instanceof UnauthorizedError) {
                System.err.println("Unauthorized error while publishing to topic: "
 + topic);
            } else {
                System.err.println("Exception occurred when using IPC.");
            }
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static PublishToTopicResponse publishBinaryMessageToTopic(
            GreengrassCoreIPCClientV2 ipcClient, String topic, String message)
 throws InterruptedException {
```

```
        BinaryMessage binaryMessage =
                new
 BinaryMessage().withMessage(message.getBytes(StandardCharsets.UTF_8));
        PublishMessage publishMessage = new
 PublishMessage().withBinaryMessage(binaryMessage);
        PublishToTopicRequest publishToTopicRequest =
                new
 PublishToTopicRequest().withTopic(topic).withPublishMessage(publishMessage);
        return ipcClient.publishToTopic(publishToTopicRequest);
    }
}
```

Python (IPC client V2)

Example 例: バイナリメッセージを発行する

```
import sys
import traceback

from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2
from awsiot.greengrasscoreipc.model import (
    PublishMessage,
    BinaryMessage
)


def main():
    args = sys.argv[1:]
    topic = args[0]
    message = args[1]

    try:
        ipc_client = GreengrassCoreIPCClientV2()
        publish_binary_message_to_topic(ipc_client, topic, message)
        print('Successfully published to topic: ' + topic)
    except Exception:
        print('Exception occurred', file=sys.stderr)
        traceback.print_exc()
        exit(1)


def publish_binary_message_to_topic(ipc_client, topic, message):
    binary_message = BinaryMessage(message=bytes(message, 'utf-8'))
    publish_message = PublishMessage(binary_message=binary_message)
```

```
    return ipc_client.publish_to_topic(topic=topic,
 publish_message=publish_message)


if __name__ == '__main__':
    main()
```

C++

Example 例: バイナリメッセージを発行する

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }

    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }

    bool OnErrorCallback(RpcError error) override {
        // Handle IPC service connection error.
        return true;
    }
};

int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
```

```
        std::cerr << "Failed to establish IPC connection: " <<
connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    String topic("my/topic");
    String message("Hello, World!");
    int timeout = 10;

    PublishToTopicRequest request;
    Vector<uint8_t> messageData({message.begin(), message.end()});
    BinaryMessage binaryMessage;
    binaryMessage.SetMessage(messageData);
    PublishMessage publishMessage;
    publishMessage.SetBinaryMessage(binaryMessage);
    request.SetTopic(topic);
    request.SetPublishMessage(publishMessage);

    auto operation = ipcClient.NewPublishToTopic();
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (!response) {
        // Handle error.
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            (void)error;
            // Handle operation error.
        } else {
            // Handle RPC error.
        }
    }
    return 0;
```

```
}
```

JavaScript

Example 例: バイナリメッセージを発行する

```
import * as greengrasscoreipc from "aws-iot-device-sdk-v2/dist/greengrasscoreipc";
import {BinaryMessage, PublishMessage, PublishToTopicRequest} from "aws-iot-device-
sdk-v2/dist/greengrasscoreipc/model";

class PublishToTopic {
    private ipcClient : greengrasscoreipc.Client
    private readonly topic : string;
    private readonly messageString : string;

    constructor() {
        // define your own constructor, e.g.
        this.topic = "<define_your_topic>";
        this.messageString = "<define_your_message_string>";
        this.publishToTopic().then(r => console.log("Started workflow"));
    }

    private async publishToTopic() {
        try {
            this.ipcClient = await getIpcClient();

            const binaryMessage : BinaryMessage = {
                message: this.messageString
            }

            const publishMessage : PublishMessage = {
                binaryMessage: binaryMessage
            }

            const request : PublishToTopicRequest = {
                topic: this.topic,
                publishMessage: publishMessage
            }

            this.ipcClient.publishToTopic(request).finally(() =>
 console.log(`Published message ${publishMessage.binaryMessage?.message} to topic`))
```

```
        } catch (e) {
            // parse the error depending on your use cases
            throw e
        }
    }
}


export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}

// starting point
const publishToTopic = new PublishToTopic();
```

SubscribeToTopic

トピックのメッセージをサブスクライブする。

このオペレーションはサブスクリプションオペレーションで、イベントメッセージのストリームをサブスクライブするというものです。このオペレーションを使用するには、イベントメッセージ、エラー、およびストリームクロージャを処理する関数を使用して、ストリームレスポンスハンドラーを定義します。(詳細については、IPC イベントストリームへのサブスクライブ を参照してください)。

イベントメッセージの種類: `SubscriptionResponseMessage`

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`topic`

サブスクライブ先のトピック。

> **Note**
>
> Greengrass nucleus v2.6.0 以降では、このトピックは MQTT トピックのワイルドカード (# と +) をサポートします。

`receiveMode` (Python: `receive_mode`)

(オプション) コンポーネントが自身からのメッセージを受信するかどうかを指定する動作。この動作を変更して、コンポーネントが自身のメッセージに基づいて動作できるようにすることができます。デフォルトの動作は、トピックに MQTT ワイルドカードが含まれているかどうかによって異なります。次のオプションから選択します。

- `RECEIVE_ALL_MESSAGES` – サブスクライブするコンポーネントからのメッセージを含む、トピックに一致するすべてのメッセージを受信します。

  このモードは、MQTT ワイルドカードを含まないトピックをサブスクライブする場合のデフォルトオプションです。

- `RECEIVE_MESSAGES_FROM_OTHERS` – サブスクライブするコンポーネントからのメッセージを除き、トピックに一致するすべてのメッセージを受信します。

  このモードは、MQTT ワイルドカードを含むトピックをサブスクライブする場合のデフォルトオプションです。

この機能は、Greengrass nucleus コンポーネントの v2.6.0 以降で利用できます。次の表に、受信モードの設定に使用する必要がある AWS IoT Device SDK の最小バージョンを示します。

SDK	最小バージョン	
AWS IoT Device SDK for Java v2	v1.9.3	
AWS IoT Device SDK for Python v2	v1.11.3	

SDK	最小バージョン
AWS IoT Device SDK for C++ v2	v1.18.4
AWS IoT Device SDK JavaScript v2 用	v1.12.0

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`messages`

メッセージのストリーム。このオブジェクト (`SubscriptionResponseMessage`) には、次の情報が含まれます。各メッセージには `jsonMessage` または `binaryMessage` が含まれます。

`jsonMessage` (Python: `json_message`)

(オプション) JSON メッセージ。このオブジェクト (`JsonMessage`) には、次の情報が含まれます。

`message`

オブジェクトとしての JSON メッセージ。

`context`

メッセージが公開されたトピックなど、メッセージのコンテキスト。

この機能は、Greengrass nucleus コンポーネントの v2.6.0 以降で利用できます。次の表に、メッセージコンテキストへのアクセスに使用する必要がある AWS IoT Device SDK の最小バージョンを示します。

SDK	最小バージョン
AWS IoT Device SDK for Java v2	v1.9.3
AWS IoT Device SDK for Python v2	v1.11.3

SDK	最小バージョン
AWS IoT Device SDK for C++ v2	v1.18.4
AWS IoT Device SDK for JavaScript v2	v1.12.0

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアは PublishToTopic および SubscribeToTopic オペレーションで同じメッセージオブジェクトを使用します。AWS IoT Greengrass Core ソフトウェアは、サブスクライブ時にこのコンテキストオブジェクトをメッセージに設定し、発行するメッセージ内のこのコンテキストオブジェクトを無視します。

このオブジェクト (MessageContext) には、次の情報が含まれます。

topic

メッセージが発行されたトピック。

binaryMessage (Python: binary_message)

(オプション) バイナリメッセージ。このオブジェクト (BinaryMessage) には、次の情報が含まれます。

message

BLOB としてのバイナリメッセージ。

context

メッセージが公開されたトピックなど、メッセージのコンテキスト。

この機能は、Greengrass nucleus コンポーネントの v2.6.0 以降で利用できます。次の表に、メッセージコンテキストへのアクセスに使用する必要がある AWS IoT Device SDK の最小バージョンを示します。

SDK	最小バージョン	
AWS IoT Device SDK for Java v2	v1.9.3	
AWS IoT Device SDK for Python v2	v1.11.3	
AWS IoT Device SDK for C++ v2	v1.18.4	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアは `PublishToTopic` および `SubscribeToTopic` オペレーションで同じメッセージオブジェクトを使用します。AWS IoT Greengrass Core ソフトウェアは、サブスクライブ時にこのコンテキストオブジェクトをメッセージに設定し、発行するメッセージ内のこのコンテキストオブジェクトを無視します。

このオブジェクト (`MessageContext`) には、次の情報が含まれます。

`topic`

メッセージが発行されたトピック。

`topicName` (Python: `topic_name`)

メッセージが発行されたトピック。

> **Note**
>
> このプロパティは現在使用されていません。Greengrass nucleus v2.6.0 以降では、`SubscriptionResponseMessage` からの `(jsonMessage|binaryMessage).context.topic` の値を取得して、メッセージが発行されたトピックを取得できます。

例

以下の例では、カスタムコンポーネントコードでこのオペレーションを呼び出す方法を示します。

Java (IPC client V2)

Example 例: ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブする

```
package com.aws.greengrass.docs.samples.ipc;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientV2;
import software.amazon.awssdk.aws.greengrass.SubscribeToTopicResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.*;

import java.nio.charset.StandardCharsets;
import java.util.Optional;

public class SubscribeToTopicV2 {

    public static void main(String[] args) {
        String topic = args[0];
        try (GreengrassCoreIPCClientV2 ipcClient =
 GreengrassCoreIPCClientV2.builder().build()) {
            SubscribeToTopicRequest request = new
 SubscribeToTopicRequest().withTopic(topic);
            GreengrassCoreIPCClientV2.StreamingResponse<SubscribeToTopicResponse,
                    SubscribeToTopicResponseHandler> response =
                    ipcClient.subscribeToTopic(request,
 SubscribeToTopicV2::onStreamEvent,
                            Optional.of(SubscribeToTopicV2::onStreamError),
                            Optional.of(SubscribeToTopicV2::onStreamClosed));
            SubscribeToTopicResponseHandler responseHandler =
 response.getHandler();
            System.out.println("Successfully subscribed to topic: " + topic);

            // Keep the main thread alive, or the process will exit.
            try {
                while (true) {
                    Thread.sleep(10000);
                }
            } catch (InterruptedException e) {
                System.out.println("Subscribe interrupted.");
            }
```

```
            // To stop subscribing, close the stream.
            responseHandler.closeStream();
        } catch (Exception e) {
            if (e.getCause() instanceof UnauthorizedError) {
                System.err.println("Unauthorized error while publishing to topic: "
 + topic);
            } else {
                System.err.println("Exception occurred when using IPC.");
            }
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static void onStreamEvent(SubscriptionResponseMessage
 subscriptionResponseMessage) {
        try {
            BinaryMessage binaryMessage =
 subscriptionResponseMessage.getBinaryMessage();
            String message = new String(binaryMessage.getMessage(),
 StandardCharsets.UTF_8);
            String topic = binaryMessage.getContext().getTopic();
            System.out.printf("Received new message on topic %s: %s%n", topic,
 message);
        } catch (Exception e) {
            System.err.println("Exception occurred while processing subscription
 response " +
                    "message.");
            e.printStackTrace();
        }
    }

    public static boolean onStreamError(Throwable error) {
        System.err.println("Received a stream error.");
        error.printStackTrace();
        return false; // Return true to close stream, false to keep stream open.
    }

    public static void onStreamClosed() {
        System.out.println("Subscribe to topic stream closed.");
    }
}
```

Python (IPC client V2)

Example 例: ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブする

```
import sys
import time
import traceback

from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2
from awsiot.greengrasscoreipc.model import (
    SubscriptionResponseMessage,
    UnauthorizedError
)


def main():
    args = sys.argv[1:]
    topic = args[0]

    try:
        ipc_client = GreengrassCoreIPCClientV2()
        # Subscription operations return a tuple with the response and the
 operation.
        _, operation = ipc_client.subscribe_to_topic(topic=topic,
 on_stream_event=on_stream_event,

 on_stream_error=on_stream_error, on_stream_closed=on_stream_closed)
        print('Successfully subscribed to topic: ' + topic)

        # Keep the main thread alive, or the process will exit.
        try:
            while True:
                time.sleep(10)
        except InterruptedError:
            print('Subscribe interrupted.')

        # To stop subscribing, close the stream.
        operation.close()
    except UnauthorizedError:
        print('Unauthorized error while subscribing to topic: ' +
              topic, file=sys.stderr)
        traceback.print_exc()
        exit(1)
    except Exception:
```

```
        print('Exception occurred', file=sys.stderr)
        traceback.print_exc()
        exit(1)


def on_stream_event(event: SubscriptionResponseMessage) -> None:
    try:
        message = str(event.binary_message.message, 'utf-8')
        topic = event.binary_message.context.topic
        print('Received new message on topic %s: %s' % (topic, message))
    except:
        traceback.print_exc()


def on_stream_error(error: Exception) -> bool:
    print('Received a stream error.', file=sys.stderr)
    traceback.print_exc()
    return False  # Return True to close stream, False to keep stream open.


def on_stream_closed() -> None:
    print('Subscribe to topic stream closed.')


if __name__ == '__main__':
    main()
```

C++

Example 例: ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブする

```
#include <iostream>

#include </crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class SubscribeResponseHandler : public SubscribeToTopicStreamHandler {
    public:
        virtual ~SubscribeResponseHandler() {}

    private:
```

```
        void OnStreamEvent(SubscriptionResponseMessage *response) override {
            auto jsonMessage = response->GetJsonMessage();
            if (jsonMessage.has_value() &&
 jsonMessage.value().GetMessage().has_value()) {
                auto messageString =
 jsonMessage.value().GetMessage().value().View().WriteReadable();
                // Handle JSON message.
            } else {
                auto binaryMessage = response->GetBinaryMessage();
                if (binaryMessage.has_value() &&
 binaryMessage.value().GetMessage().has_value()) {
                    auto messageBytes = binaryMessage.value().GetMessage().value();
                    std::string messageString(messageBytes.begin(),
 messageBytes.end());
                    // Handle binary message.
                }
            }
        }

        bool OnStreamError(OperationError *error) override {
            // Handle error.
            return false; // Return true to close stream, false to keep stream open.
        }

        void OnStreamClosed() override {
            // Handle close.
        }
};

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }

    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }

    bool OnErrorCallback(RpcError error) override {
        // Handle IPC service connection error.
        return true;
    }
};
```

```
int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
 connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    String topic("my/topic");
    int timeout = 10;

    SubscribeToTopicRequest request;
    request.SetTopic(topic);

    //SubscribeResponseHandler streamHandler;
    auto streamHandler = MakeShared<SubscribeResponseHandler>(DefaultAllocator());
    auto operation = ipcClient.NewSubscribeToTopic(streamHandler);
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
 std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
 Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (!response) {
        // Handle error.
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            (void)error;
            // Handle operation error.
        } else {
            // Handle RPC error.
```

```
        }
        exit(-1);
    }

    // Keep the main thread alive, or the process will exit.
    while (true) {
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    operation->Close();
    return 0;
}
```

JavaScript

Example 例: ローカルのパブリッシュ/サブスクライブメッセージをサブスクライブする

```
import * as greengrasscoreipc from "aws-iot-device-sdk-v2/dist/greengrasscoreipc";
import {SubscribeToTopicRequest, SubscriptionResponseMessage} from "aws-iot-device-
sdk-v2/dist/greengrasscoreipc/model";
import {RpcError} from "aws-iot-device-sdk-v2/dist/eventstream_rpc";

class SubscribeToTopic {
    private ipcClient : greengrasscoreipc.Client
    private readonly topic : string;

    constructor() {
        // define your own constructor, e.g.
        this.topic = "<define_your_topic>";
        this.subscribeToTopic().then(r => console.log("Started workflow"));
    }

    private async subscribeToTopic() {
        try {
            this.ipcClient = await getIpcClient();

            const subscribeToTopicRequest : SubscribeToTopicRequest = {
                topic: this.topic,
            }

            const streamingOperation =
 this.ipcClient.subscribeToTopic(subscribeToTopicRequest, undefined); //
 conditionally apply options
```

```
            streamingOperation.on("message", (message: SubscriptionResponseMessage)
 => {
                // parse the message depending on your use cases, e.g.
                if(message.binaryMessage && message.binaryMessage.message) {
                    const receivedMessage =
 message.binaryMessage?.message.toString();
                }
            });

            streamingOperation.on("streamError", (error : RpcError) => {
                // define your own error handling logic
            })

            streamingOperation.on("ended", () => {
                // define your own logic
            })

            await streamingOperation.activate();

            // Keep the main thread alive, or the process will exit.
            await new Promise((resolve) => setTimeout(resolve, 10000))
        } catch (e) {
            // parse the error depending on your use cases
            throw e
        }
    }
}

export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}
```

```
// starting point
const subscribeToTopic = new SubscribeToTopic();
```

例

コンポーネントのパブリッシュ/サブスクライブ IPC サービスの使用方法については、以下の例を参照してください。

パブリッシュ/サブスクライブパブリッシャー (Java、IPC クライアント V1) の例

以下の recipe の例は、コンポーネントをすべてのトピックに発行できるようにします。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubPublisherJava",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubPublisherJava:pubsub:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "run": "java -jar {artifacts:path}/PubSubPublisher.jar"
      }
```

```
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubPublisherJava
ComponentVersion: '1.0.0'
ComponentDescription: A component that publishes messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        'com.example.PubSubPublisherJava:pubsub:1':
          policyDescription: Allows access to publish to all topics.
          operations:
            - 'aws.greengrass#PublishToTopic'
          resources:
            - '*'
Manifests:
  - Lifecycle:
      run: |-
        java -jar {artifacts:path}/PubSubPublisher.jar
```

以下の Java アプリケーションの例は、パブリッシュ/サブスクライブ IPC サービスを使用して、他コンポーネントにメッセージを発行する方法を示します。

```
/* Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0 */

package com.example.ipc.pubsub;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.model.*;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;

import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
```

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class PubSubPublisher {

    public static void main(String[] args) {
        String message = "Hello from the pub/sub publisher (Java).";
        String topic = "test/topic/java";

        try (EventStreamRPCConnection eventStreamRPCConnection =
 IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient = new
 GreengrassCoreIPCClient(eventStreamRPCConnection);

            while (true) {
                PublishToTopicRequest publishRequest = new PublishToTopicRequest();
                PublishMessage publishMessage = new PublishMessage();
                BinaryMessage binaryMessage = new BinaryMessage();
                binaryMessage.setMessage(message.getBytes(StandardCharsets.UTF_8));
                publishMessage.setBinaryMessage(binaryMessage);
                publishRequest.setPublishMessage(publishMessage);
                publishRequest.setTopic(topic);
                CompletableFuture<PublishToTopicResponse> futureResponse = ipcClient
                        .publishToTopic(publishRequest,
 Optional.empty()).getResponse();

                try {
                    futureResponse.get(10, TimeUnit.SECONDS);
                    System.out.println("Successfully published to topic: " + topic);
                } catch (TimeoutException e) {
                    System.err.println("Timeout occurred while publishing to topic: " +
 topic);
                } catch (ExecutionException e) {
                    if (e.getCause() instanceof UnauthorizedError) {
                        System.err.println("Unauthorized error while publishing to
 topic: " + topic);
                    } else {
                        System.err.println("Execution exception while publishing to
 topic: " + topic);
                    }
                    throw e;
                }
                Thread.sleep(5000);
```

```
            }
        } catch (InterruptedException e) {
            System.out.println("Publisher interrupted.");
        } catch (Exception e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

パブリッシュ/サブスクライブサブスクライバー (Java、IPC クライアント V1) の例

以下の recipe の例は、コンポーネントをすべてのトピックをサブスクライブできるようにします。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubSubscriberJava",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubSubscriberJava:pubsub:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "run": "java -jar {artifacts:path}/PubSubSubscriber.jar"
```

```
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubSubscriberJava
ComponentVersion: '1.0.0'
ComponentDescription: A component that subscribes to messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        'com.example.PubSubSubscriberJava:pubsub:1':
          policyDescription: Allows access to subscribe to all topics.
          operations:
            - 'aws.greengrass#SubscribeToTopic'
          resources:
            - '*'
Manifests:
  - Lifecycle:
      run: |-
        java -jar {artifacts:path}/PubSubSubscriber.jar
```

以下の Java アプリケーションの例は、パブリッシュ/サブスクライブ IPC サービスを使用して、他コンポーネントのメッセージをサブスクライブする方法を示します。

```
/* Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0 */

package com.example.ipc.pubsub;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.SubscribeToTopicResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.SubscribeToTopicRequest;
import software.amazon.awssdk.aws.greengrass.model.SubscribeToTopicResponse;
import software.amazon.awssdk.aws.greengrass.model.SubscriptionResponseMessage;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
```

```
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;
import software.amazon.awssdk.eventstreamrpc.StreamResponseHandler;

import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class PubSubSubscriber {

    public static void main(String[] args) {
        String topic = "test/topic/java";

        try (EventStreamRPCConnection eventStreamRPCConnection =
 IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient = new
 GreengrassCoreIPCClient(eventStreamRPCConnection);

            SubscribeToTopicRequest subscribeRequest = new SubscribeToTopicRequest();
            subscribeRequest.setTopic(topic);
            SubscribeToTopicResponseHandler operationResponseHandler = ipcClient
                    .subscribeToTopic(subscribeRequest, Optional.of(new
 SubscribeResponseHandler()));
            CompletableFuture<SubscribeToTopicResponse> futureResponse =
 operationResponseHandler.getResponse();

            try {
                futureResponse.get(10, TimeUnit.SECONDS);
                System.out.println("Successfully subscribed to topic: " + topic);
            } catch (TimeoutException e) {
                System.err.println("Timeout occurred while subscribing to topic: " +
 topic);
                throw e;
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.println("Unauthorized error while subscribing to topic:
 " + topic);
                } else {
                    System.err.println("Execution exception while subscribing to topic:
 " + topic);
                }
                throw e;
```

```
            }

            // Keep the main thread alive, or the process will exit.
            try {
                while (true) {
                    Thread.sleep(10000);
                }
            } catch (InterruptedException e) {
                System.out.println("Subscribe interrupted.");
            }
        } catch (Exception e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    private static class SubscribeResponseHandler implements
StreamResponseHandler<SubscriptionResponseMessage> {

        @Override
        public void onStreamEvent(SubscriptionResponseMessage
subscriptionResponseMessage) {
            try {
                String message = new
String(subscriptionResponseMessage.getBinaryMessage()
                        .getMessage(), StandardCharsets.UTF_8);
                System.out.println("Received new message: " + message);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        @Override
        public boolean onStreamError(Throwable error) {
            System.err.println("Received a stream error.");
            error.printStackTrace();
            return false; // Return true to close stream, false to keep stream open.
        }

        @Override
        public void onStreamClosed() {
            System.out.println("Subscribe to topic stream closed.");
        }
```

```
    }
}
```

パブリッシュ/サブスクライブパブリッシャー (Python、IPC クライアント V1) の例

以下の recipe の例は、コンポーネントをすべてのトピックに発行できるようにします。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubPublisherPython",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubPublisherPython:pubsub:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "python3 -m pip install --user awsiotsdk",
        "run": "python3 -u {artifacts:path}/pubsub_publisher.py"
      }
    },
    {
      "Platform": {
```

```
        "os": "windows"
      },
      "Lifecycle": {
        "install": "py -3 -m pip install --user awsiotsdk",
        "run": "py -3 -u {artifacts:path}/pubsub_publisher.py"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubPublisherPython
ComponentVersion: 1.0.0
ComponentDescription: A component that publishes messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        com.example.PubSubPublisherPython:pubsub:1:
          policyDescription: Allows access to publish to all topics.
          operations:
            - aws.greengrass#PublishToTopic
          resources:
            - "*"
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: python3 -m pip install --user awsiotsdk
      run: python3 -u {artifacts:path}/pubsub_publisher.py
  - Platform:
      os: windows
    Lifecycle:
      install: py -3 -m pip install --user awsiotsdk
      run: py -3 -u {artifacts:path}/pubsub_publisher.py
```

以下の Python アプリケーションの例は、パブリッシュ/サブスクライブ IPC サービスを使用して、他コンポーネントにメッセージを発行する方法を示します。

```
import concurrent.futures
import sys
import time
import traceback

import awsiot.greengrasscoreipc
from awsiot.greengrasscoreipc.model import (
    PublishToTopicRequest,
    PublishMessage,
    BinaryMessage,
    UnauthorizedError
)


topic = "test/topic/python"
message = "Hello from the pub/sub publisher (Python)."
TIMEOUT = 10

try:
    ipc_client = awsiot.greengrasscoreipc.connect()

    while True:
        request = PublishToTopicRequest()
        request.topic = topic
        publish_message = PublishMessage()
        publish_message.binary_message = BinaryMessage()
        publish_message.binary_message.message = bytes(message, "utf-8")
        request.publish_message = publish_message
        operation = ipc_client.new_publish_to_topic()
        operation.activate(request)
        future_response = operation.get_response()

        try:
            future_response.result(TIMEOUT)
            print('Successfully published to topic: ' + topic)
        except concurrent.futures.TimeoutError:
            print('Timeout occurred while publishing to topic: ' + topic,
 file=sys.stderr)
       except UnauthorizedError as e:
            print('Unauthorized error while publishing to topic: ' + topic,
 file=sys.stderr)
            raise e
       except Exception as e:
```

```
            print('Exception while publishing to topic: ' + topic, file=sys.stderr)
            raise e
        time.sleep(5)
except InterruptedError:
    print('Publisher interrupted.')
except Exception:
    print('Exception occurred when using IPC.', file=sys.stderr)
    traceback.print_exc()
    exit(1)
```

パブリッシュ/サブスクライブサブスクライバー (Python、IPC クライアント V1) の例

以下の recipe の例は、コンポーネントをすべてのトピックをサブスクライブできるようにします。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubSubscriberPython",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubSubscriberPython:pubsub:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
```

```
      "Lifecycle": {
        "install": "python3 -m pip install --user awsiotsdk",
        "run": "python3 -u {artifacts:path}/pubsub_subscriber.py"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "py -3 -m pip install --user awsiotsdk",
        "run": "py -3 -u {artifacts:path}/pubsub_subscriber.py"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubSubscriberPython
ComponentVersion: 1.0.0
ComponentDescription: A component that subscribes to messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        com.example.PubSubSubscriberPython:pubsub:1:
          policyDescription: Allows access to subscribe to all topics.
          operations:
            - aws.greengrass#SubscribeToTopic
          resources:
            - "*"
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: python3 -m pip install --user awsiotsdk
      run: python3 -u {artifacts:path}/pubsub_subscriber.py
  - Platform:
      os: windows
```

```
    Lifecycle:
      install: py -3 -m pip install --user awsiotsdk
      run: py -3 -u {artifacts:path}/pubsub_subscriber.py
```

以下の Python アプリケーションの例は、パブリッシュ/サブスクライブ IPC サービスを使用して、他コンポーネントのメッセージをサブスクライブする方法を示します。

```
import concurrent.futures
import sys
import time
import traceback

import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import (
    SubscribeToTopicRequest,
    SubscriptionResponseMessage,
    UnauthorizedError
)

topic = "test/topic/python"
TIMEOUT = 10


class StreamHandler(client.SubscribeToTopicStreamHandler):
    def __init__(self):
        super().__init__()

    def on_stream_event(self, event: SubscriptionResponseMessage) -> None:
        try:
            message = str(event.binary_message.message, "utf-8")
            print("Received new message: " + message)
        except:
            traceback.print_exc()

    def on_stream_error(self, error: Exception) -> bool:
        print("Received a stream error.", file=sys.stderr)
        traceback.print_exc()
        return False  # Return True to close stream, False to keep stream open.

    def on_stream_closed(self) -> None:
        print('Subscribe to topic stream closed.')
```

```
try:
    ipc_client = awsiot.greengrasscoreipc.connect()

    request = SubscribeToTopicRequest()
    request.topic = topic
    handler = StreamHandler()
    operation = ipc_client.new_subscribe_to_topic(handler)
    operation.activate(request)
    future_response = operation.get_response()

    try:
        future_response.result(TIMEOUT)
        print('Successfully subscribed to topic: ' + topic)
    except concurrent.futures.TimeoutError as e:
        print('Timeout occurred while subscribing to topic: ' + topic,
 file=sys.stderr)
        raise e
    except UnauthorizedError as e:
        print('Unauthorized error while subscribing to topic: ' + topic,
 file=sys.stderr)
        raise e
    except Exception as e:
        print('Exception while subscribing to topic: ' + topic, file=sys.stderr)
        raise e

    # Keep the main thread alive, or the process will exit.
    try:
        while True:
            time.sleep(10)
    except InterruptedError:
        print('Subscribe interrupted.')
except Exception:
    print('Exception occurred when using IPC.', file=sys.stderr)
    traceback.print_exc()
    exit(1)
```

パブリッシュ/サブスクライブパブリッシャー (C++) の例

以下の recipe の例は、コンポーネントをすべてのトピックに発行できるようにします。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubPublisherCpp",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubPublisherCpp:pubsub:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "run": "{artifacts:path}/greengrassv2_pubsub_publisher"
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.PubSubPublisherCpp/1.0.0/greengrassv2_pubsub_publisher",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubPublisherCpp
ComponentVersion: 1.0.0
ComponentDescription: A component that publishes messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        com.example.PubSubPublisherCpp:pubsub:1:
          policyDescription: Allows access to publish to all topics.
          operations:
            - aws.greengrass#PublishToTopic
          resources:
            - "*"
Manifests:
  - Lifecycle:
      run: "{artifacts:path}/greengrassv2_pubsub_publisher"
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.PubSubPublisherCpp/1.0.0/
greengrassv2_pubsub_publisher
        Permission:
          Execute: OWNER
```

以下の C++ アプリケーションの例は、パブリッシュ/サブスクライブ IPC サービスを使用して、他コンポーネントにメッセージを発行する方法を示します。

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
    }
```

```
    void OnDisconnectCallback(RpcError error) override {
        std::cout << "OnDisconnectCallback: " << error.StatusToString() << std::endl;
        exit(-1);
    }

    bool OnErrorCallback(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
};

int main() {
    String message("Hello from the pub/sub publisher (C++).");
    String topic("test/topic/cpp");
    int timeout = 10;

    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
 connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    while (true) {
        PublishToTopicRequest request;
        Vector<uint8_t> messageData({message.begin(), message.end()});
        BinaryMessage binaryMessage;
        binaryMessage.SetMessage(messageData);
        PublishMessage publishMessage;
        publishMessage.SetBinaryMessage(binaryMessage);
        request.SetTopic(topic);
        request.SetPublishMessage(publishMessage);

        auto operation = ipcClient.NewPublishToTopic();
        auto activate = operation->Activate(request, nullptr);
        activate.wait();
```

```
        auto responseFuture = operation->GetResult();
        if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
 std::future_status::timeout) {
            std::cerr << "Operation timed out while waiting for response from
 Greengrass Core." << std::endl;
            exit(-1);
        }

        auto response = responseFuture.get();
        if (response) {
            std::cout << "Successfully published to topic: " << topic << std::endl;
        } else {
            // An error occurred.
            std::cout << "Failed to publish to topic: " << topic << std::endl;
            auto errorType = response.GetResultType();
            if (errorType == OPERATION_ERROR) {
                auto *error = response.GetOperationError();
                std::cout << "Operation error: " << error->GetMessage().value() <<
 std::endl;
            } else {
                std::cout << "RPC error: " << response.GetRpcError() << std::endl;
            }
            exit(-1);
        }

        std::this_thread::sleep_for(std::chrono::seconds(5));
    }

    return 0;
}
```

パブリッシュ/サブスクライブサブスクライバー (C++) の例

以下の recipe の例は、コンポーネントをすべてのトピックをサブスクライブできるようにします。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubSubscriberCpp",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
```

```
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubSubscriberCpp:pubsub:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "run": "{artifacts:path}/greengrassv2_pub_sub_subscriber"
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.PubSubSubscriberCpp/1.0.0/greengrassv2_pub_sub_subscriber",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubSubscriberCpp
ComponentVersion: 1.0.0
ComponentDescription: A component that subscribes to messages.
ComponentPublisher: Amazon
ComponentConfiguration:
```

```
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        com.example.PubSubSubscriberCpp:pubsub:1:
          policyDescription: Allows access to subscribe to all topics.
          operations:
            - aws.greengrass#SubscribeToTopic
          resources:
            - "*"
Manifests:
  - Lifecycle:
      run: "{artifacts:path}/greengrassv2_pub_sub_subscriber"
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.PubSubSubscriberCpp/1.0.0/greengrassv2_pub_sub_subscriber
        Permission:
          Execute: OWNER
```

以下の C++ アプリケーションの例は、パブリッシュ/サブスクライブ IPC サービスを使用して、他コンポーネントのメッセージをサブスクライブする方法を示します。

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class SubscribeResponseHandler : public SubscribeToTopicStreamHandler {
    public:
        virtual ~SubscribeResponseHandler() {}

    private:
        void OnStreamEvent(SubscriptionResponseMessage *response) override {
            auto jsonMessage = response->GetJsonMessage();
            if (jsonMessage.has_value() &&
 jsonMessage.value().GetMessage().has_value()) {
                auto messageString =
 jsonMessage.value().GetMessage().value().View().WriteReadable();
                std::cout << "Received new message: " << messageString << std::endl;
            } else {
```

```
                auto binaryMessage = response->GetBinaryMessage();
                if (binaryMessage.has_value() &&
 binaryMessage.value().GetMessage().has_value()) {
                    auto messageBytes = binaryMessage.value().GetMessage().value();
                    std::string messageString(messageBytes.begin(),
 messageBytes.end());
                    std::cout << "Received new message: " << messageString <<
 std::endl;
                }
            }
        }

        bool OnStreamError(OperationError *error) override {
            std::cout << "Received an operation error: ";
            if (error->GetMessage().has_value()) {
                std::cout << error->GetMessage().value();
            }
            std::cout << std::endl;
            return false; // Return true to close stream, false to keep stream open.
        }

        void OnStreamClosed() override {
            std::cout << "Subscribe to topic stream closed." << std::endl;
        }
};

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
    }

    void OnDisconnectCallback(RpcError error) override {
        std::cout << "OnDisconnectCallback: " << error.StatusToString() << std::endl;
        exit(-1);
    }

    bool OnErrorCallback(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
};

int main() {
    String topic("test/topic/cpp");
```

```
    int timeout = 10;

    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    SubscribeToTopicRequest request;
    request.SetTopic(topic);
    auto streamHandler = MakeShared<SubscribeResponseHandler>(DefaultAllocator());
    auto operation = ipcClient.NewSubscribeToTopic(streamHandler);
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (response) {
        std::cout << "Successfully subscribed to topic: " << topic << std::endl;
    } else {
        // An error occurred.
        std::cout << "Failed to subscribe to topic: " << topic << std::endl;
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            std::cout << "Operation error: " << error->GetMessage().value() <<
std::endl;
        } else {
            std::cout << "RPC error: " << response.GetRpcError() << std::endl;
        }
```

```
        exit(-1);
    }

    // Keep the main thread alive, or the process will exit.
    while (true) {
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    operation->Close();
    return 0;
}
```

AWS IoT Core MQTT メッセージをパブリッシュ/サブスクライブする

AWS IoT Core MQTT メッセージング IPC サービスを使用すると、AWS IoT Core と MQTT メッセージの送受信を行えます。コンポーネントは AWS IoT Core にメッセージをパブリッシュできるとともに、トピックをサブスクライブして、他のソースからの MQTT メッセージに対応できます。AWS IoT Core の MQTT の実装の詳細については、「AWS IoT Core デベロッパーガイド」の「MQTT」を参照してください。

> **Note**
>
> この MQTT メッセージング IPC サービスを使用すると、AWS IoT Core とメッセージを交換できるようになります。コンポーネント間でメッセージを交換する方法の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。

トピック

最小 SDK バージョン

以下の表に、AWS IoT Core と MQTT メッセージの発行およびサブスクライブのやりとりを行う際に使用が必要となる AWS IoT Device SDK の最小バージョンを示します。

SDK	最小バージョン	
AWS IoT Device SDK for Java v2	v1.2.10	
AWS IoT Device SDK for Python v2	v1.5.3	
AWS IoT Device SDK for C++ v2	v1.17.0	
AWS IoT Device SDK JavaScript v2 用	v1.12.0	

認証

AWS IoT Core MQTT メッセージングをカスタムコンポーネントで使用するには、コンポーネントでトピックに関するメッセージを送受信できるようにする承認ポリシーを定義する必要があります。承認ポリシーの定義については、「コンポーネントに IPC オペレーションの実行を許可する」を参照してください。

AWS IoT Core MQTT メッセージングの承認ポリシーには以下のプロパティがあります。

IPC サービス識別子: `aws.greengrass.ipc.mqttproxy`

操作	説明	リソース
`aws.greengrass#PublishToIoTCore`	コンポーネントが、指定した MQTT トピックの AWS IoT Core にメッセージを発行できるようにします。	`test/topic` などのトピック文字列、または `*` ですべてのトピックへのアクセスを許可します。MQTT トピックのワイルドカード (# および

操作	説明	リソース
		+) を使用して、複数のリソースを一致させることができます。
aws.greengrass#SubscribeToIoTCore	コンポーネントが、指定したトピックの AWS IoT Core からのメッセージをサブスクライブできるようにします。	test/topic などのトピック文字列、または * ですべてのトピックへのアクセスを許可します。MQTT トピックのワイルドカード (# および +) を使用して、複数のリソースを一致させることができます。
*	コンポーネントが、指定したトピックの AWS IoT Core MQTT メッセージを発行およびサブスクライブできるようにします。	test/topic などのトピック文字列、または * ですべてのトピックへのアクセスを許可します。MQTT トピックのワイルドカード (# および +) を使用して、複数のリソースを一致させることができます。

AWS IoT Core MQTT 承認ポリシーの MQTT ワイルドカード

AWS IoT Core MQTT IPC 承認ポリシーで MQTT ワイルドカードを使用できます。これによりコンポーネントは、承認ポリシーで許可するトピックフィルターに一致するトピックを発行およびサブスクライブできます。例えば、コンポーネントの承認ポリシーで `test/topic/#` へのアクセス権が付与されている場合、コンポーネントは `test/topic/#` をサブスクライブでき、`test/topic/filter` を発行およびサブスクライブできます。

AWS IoT Core MQTT 承認ポリシーのレシピ変数

v2.6.0 以降の Greengrass nucleus を使用している場合、承認ポリシーの `{iot:thingName}` recipe 変数を使用できます。この機能を使用すると、コアデバイスのグループに対して 1 つの承認ポリシーを設定できます。各コアデバイスは自身の名前を含むトピックにのみアクセスできます。例えば、コンポーネントに次のトピックリソースへのアクセスを許可できます。

```
devices/{iot:thingName}/messages
```

詳細については、「レシピ変数 と マージ更新で recipe 変数を使用する」を参照してください。

承認ポリシーの例

次の承認ポリシーの例を参照して、コンポーネントの承認ポリシー設定の参考にできます。

Example アクセスが制限されていない承認ポリシーの例

以下の承認ポリシーの例では、コンポーネントがすべてのトピックを公開およびサブスクライブすることを許可します。

JSON

```
{
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
      "com.example.MyIoTCorePubSubComponent:mqttproxy:1": {
        "policyDescription": "Allows access to publish/subscribe to all topics.",
        "operations": [
          "aws.greengrass#PublishToIoTCore",
          "aws.greengrass#SubscribeToIoTCore"
        ],
        "resources": [
          "*"
        ]
      }
    }
  }
}
```

YAML

```
---
accessControl:
  aws.greengrass.ipc.mqttproxy:
    com.example.MyIoTCorePubSubComponent:mqttproxy:1:
      policyDescription: Allows access to publish/subscribe to all topics.
      operations:
        - aws.greengrass#PublishToIoTCore
```

```
        - aws.greengrass#SubscribeToIoTCore
      resources:
        - "*"
```

Example アクセスが制限されている承認ポリシーの例

次の承認ポリシーの例では、コンポーネントが、factory/1/events および factory/1/actions という名前の 2 つのトピックを公開およびサブスクライブすることを許可します。

JSON

```
{
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
      "com.example.MyIoTCorePubSubComponent:mqttproxy:1": {
        "policyDescription": "Allows access to publish/subscribe to factory 1
 topics.",
        "operations": [
          "aws.greengrass#PublishToIoTCore",
          "aws.greengrass#SubscribeToIoTCore"
        ],
        "resources": [
          "factory/1/actions",
          "factory/1/events"
        ]
      }
    }
  }
}
```

YAML

```
---
accessControl:
  aws.greengrass.ipc.mqttproxy:
    "com.example.MyIoTCorePubSubComponent:mqttproxy:1":
      policyDescription: Allows access to publish/subscribe to factory 1 topics.
      operations:
        - aws.greengrass#PublishToIoTCore
        - aws.greengrass#SubscribeToIoTCore
      resources:
        - factory/1/actions
```

```
        - factory/1/events
```

Example コアデバイスのグループに対する承認ポリシーの例

> **Important**
>
> この例では、v2.6.0 以降の Greengrass nucleus コンポーネントで利用できる機能を使用しています。Greengrass nucleus v2.6.0 では、コンポーネント設定に、ほとんどの recipe 変数 (`{iot:thingName}` など) のサポートが追加されました。

次の承認ポリシーの例は、コンポーネントが実行されているコアデバイスの名前を含むトピックを、コンポーネントが発行およびサブスクライブできるようにします。

JSON

```
{
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
      "com.example.MyIoTCorePubSubComponent:mqttproxy:1": {
        "policyDescription": "Allows access to publish/subscribe to all topics.",
        "operations": [
          "aws.greengrass#PublishToIoTCore",
          "aws.greengrass#SubscribeToIoTCore"
        ],
        "resources": [
          "factory/1/devices/{iot:thingName}/controls"
        ]
      }
    }
  }
}
```

YAML

```
---
accessControl:
  aws.greengrass.ipc.mqttproxy:
    "com.example.MyIoTCorePubSubComponent:mqttproxy:1":
      policyDescription: Allows access to publish/subscribe to all topics.
      operations:
```

```
        - aws.greengrass#PublishToIoTCore
        - aws.greengrass#SubscribeToIoTCore
      resources:
        - factory/1/devices/{iot:thingName}/controls
```

PublishToIoTCore

トピックに関して、AWS IoT Core に MQTT メッセージを発行します。

MQTT メッセージを AWS IoT Core に公開する場合、1 秒あたり 100 トランザクションのクォータがあります。このクォータを超えると、メッセージは Greengrass デバイスでの処理のためにキューに入れられます。また、1 秒あたり 512 KB のデータというクォータと、アカウント全体で 1 秒あたり 20,000 パブリッシャー (一部の AWS リージョン では 2,000 件) のクォータがあります。AWS IoT Core の MQTT メッセージブローカー制限の詳細については、「AWS IoT Core メッセージブローカーとプロトコルの制限とクォータ」を参照してください。

これらのクォータを超えると、Greengrass デバイスはメッセージの公開を AWS IoT Core に制限します。メッセージはメモリ内のスプーラに保存されます。デフォルトでは、スプーラに割り当てられるメモリは 2.5 MB です。スプーラがいっぱいになると、新しいメッセージは拒否されます。スプーラのサイズを増やすことができます。詳細については、Greengrass nucleus ドキュメントの「構成」を参照してください。スプーラがいっぱいになり、割り当てられるメモリを増やす必要がないように、公開要求は 1 秒あたり 100 要求以下に制限してください。

アプリケーションがメッセージをより高いレートで送信したり、より大きなメッセージを送信したりする必要がある場合は、ストリームマネージャー を使用して Kinesis データストリームにメッセージを送信することを検討してください。ストリームマネージャーコンポーネントは、大量のデータを AWS クラウド に転送するように設計されています。詳細については、「Greengrass コアデバイスでのデータストリームの管理」を参照してください。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`topicName` (Python: `topic_name`)

メッセージの発行先として指定するトピック。

`qos`

使用する MQTT QoS。この列挙型 (QOS) には以下の値があります。

- AT_MOST_ONCE – QoS 0。MQTT メッセージが配信されるのは 1 回以下です。
- AT_LEAST_ONCE – QoS 1。MQTT メッセージが配信されるのは 1 回以上です。

`payload`

(オプション) BLOB としてのメッセージペイロード。

MQTT 5を使用する際、Greengrass nucleus の v2.10.0 以降で以下の機能が使用できます。MQTT 3.1.1 を使用している場合、これらの機能は無視されます。これらの機能を利用するために必要な AWS IoT デバイス SDK の最小バージョンは、次の表のとおりです。

SDK	最小バージョン
AWS IoT Device SDK for Python v2	v1.15.0
AWS IoT Device SDK for Java v2	v1.13.0
AWS IoT Device SDK for C++ v2	v1.24.0
AWS IoT Device SDK for JavaScript v2	v1.13.0

`payloadFormat`

(オプション) メッセージペイロードのフォーマット。payloadFormat を設定しない場合、タイプは BYTES とみなされます。この列挙型には以下の値があります。

- BYTES— ペイロードのコンテンツは、バイナリ BLOB です。
- UTF8— ペイロードのコンテンツは UTF8 の文字列です。

`retain`

(オプション) 発行時に MQTT 保持オプションを `true` に設定するか否かを示します。

`userProperties`

(オプション) 送信するアプリケーション固有の UserProperty オブジェクトのリストです。UserProperty オブジェクトは次のように定義されます。

```
UserProperty:
  key: string
```

```
  value: string
```

`messageExpiryIntervalSeconds`

(オプション) メッセージが期限切れとなり、サーバーによって削除されるまでの秒数です。この値を設定しない場合、メッセージに有効期限は設定されません。

`correlationData`

(オプション) リクエストに付加される情報で、リクエストとレスポンスの関連付けに使用できます。

`responseTopic`

(オプション) レスポンスメッセージに使用するトピックです。

`contentType`

(オプション) メッセージのコンテンツタイプを示すアプリケーション固有の識別子です。

レスポンス

このオペレーションはレスポンスで一切の情報を提供しません。

例

以下の例では、カスタムコンポーネントコードでこのオペレーションを呼び出す方法を示します。

Java (IPC client V2)

Example 例:メッセージを発行する

```
package com.aws.greengrass.docs.samples.ipc;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientV2;
import software.amazon.awssdk.aws.greengrass.model.PublishToIoTCoreRequest;
import software.amazon.awssdk.aws.greengrass.model.QOS;
import java.nio.charset.StandardCharsets;

public class PublishToIoTCore {

    public static void main(String[] args) {
```

```
        String topic = args[0];
        String message = args[1];
        QOS qos = QOS.get(args[2]);

        try (GreengrassCoreIPCClientV2 ipcClientV2 =
 GreengrassCoreIPCClientV2.builder().build()) {
            ipcClientV2.publishToIoTCore(new PublishToIoTCoreRequest()
                    .withTopicName(topic)
                    .withPayload(message.getBytes(StandardCharsets.UTF_8))
                    .withQos(qos));
            System.out.println("Successfully published to topic: " + topic);
        } catch (Exception e) {
            System.err.println("Exception occurred.");
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Python (IPC client V2)

Example 例:メッセージを発行する

> **Note**
>
> この例では、AWS IoT Device SDK for Python v2 のバージョン 1.5.4 以降を使用していることを前提としています。

```
import awsiot.greengrasscoreipc.clientv2 as clientV2

topic = 'my/topic'
qos = '1'
payload = 'Hello, World'

ipc_client = clientV2.GreengrassCoreIPCClientV2()
resp = ipc_client.publish_to_iot_core(topic_name=topic, qos=qos, payload=payload)
ipc_client.close()
```

Java (IPC client V1)

Example 例:メッセージを発行する

> **Note**
>
> この例は `IPCUtils` クラスを使用して、AWS IoT Greengrass Core IPC サービスへの接続を作成します。(詳細については、AWS IoT Greengrass Core IPC サービスに接続するを参照してください)。

```
package com.aws.greengrass.docs.samples.ipc;

import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.PublishToIoTCoreResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.PublishToIoTCoreRequest;
import software.amazon.awssdk.aws.greengrass.model.PublishToIoTCoreResponse;
import software.amazon.awssdk.aws.greengrass.model.QOS;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;

import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class PublishToIoTCore {

    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        String topic = args[0];
        String message = args[1];
        QOS qos = QOS.get(args[2]);
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            PublishToIoTCoreResponseHandler responseHandler =
```

```
                    PublishToIoTCore.publishBinaryMessageToTopic(ipcClient, topic,
 message, qos);
            CompletableFuture<PublishToIoTCoreResponse> futureResponse =
                    responseHandler.getResponse();
            try {
                futureResponse.get(TIMEOUT_SECONDS, TimeUnit.SECONDS);
                System.out.println("Successfully published to topic: " + topic);
            } catch (TimeoutException e) {
                System.err.println("Timeout occurred while publishing to topic: " +
 topic);
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.println("Unauthorized error while publishing to
 topic: " + topic);
                } else {
                    throw e;
                }
            }
        } catch (InterruptedException e) {
            System.out.println("IPC interrupted.");
        } catch (ExecutionException e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static PublishToIoTCoreResponseHandler
 publishBinaryMessageToTopic(GreengrassCoreIPCClient greengrassCoreIPCClient, String
 topic, String message, QOS qos) {
        PublishToIoTCoreRequest publishToIoTCoreRequest = new
 PublishToIoTCoreRequest();
        publishToIoTCoreRequest.setTopicName(topic);

 publishToIoTCoreRequest.setPayload(message.getBytes(StandardCharsets.UTF_8));
        publishToIoTCoreRequest.setQos(qos);
        return greengrassCoreIPCClient.publishToIoTCore(publishToIoTCoreRequest,
 Optional.empty());
    }
}
```

Python (IPC client V1)

Example 例:メッセージを発行する

> **Note**
>
> この例では、AWS IoT Device SDK for Python v2 のバージョン 1.5.4 以降を使用していることを前提としています。

```
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import (
    QOS,
    PublishToIoTCoreRequest
)

TIMEOUT = 10

ipc_client = awsiot.greengrasscoreipc.connect()

topic = "my/topic"
message = "Hello, World"
qos = QOS.AT_LEAST_ONCE

request = PublishToIoTCoreRequest()
request.topic_name = topic
request.payload = bytes(message, "utf-8")
request.qos = qos
operation = ipc_client.new_publish_to_iot_core()
operation.activate(request)
future_response = operation.get_response()
future_response.result(TIMEOUT)
```

C++

Example 例:メッセージを発行する

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>
```

```
using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }

    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }

    bool OnErrorCallback(RpcError error) override {
        // Handle IPC service connection error.
        return true;
    }
};

int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
 connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    String message("Hello, World!");
    String topic("my/topic");
    QOS qos = QOS_AT_MOST_ONCE;
    int timeout = 10;

    PublishToIoTCoreRequest request;
    Vector<uint8_t> messageData({message.begin(), message.end()});
    request.SetTopicName(topic);
    request.SetPayload(messageData);
    request.SetQos(qos);

    auto operation = ipcClient.NewPublishToIoTCore();
```

```
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
 std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
 Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (!response) {
        // Handle error.
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            (void)error;
            // Handle operation error.
        } else {
            // Handle RPC error.
        }
    }

    return 0;
}
```

JavaScript

Example 例:メッセージを発行する

```
import * as greengrasscoreipc from "aws-iot-device-sdk-v2/dist/greengrasscoreipc";
import {QOS, SubscribeToIoTCoreRequest} from "aws-iot-device-sdk-v2/dist/
greengrasscoreipc/model";

class PublishToIoTCore {
    private ipcClient : greengrasscoreipc.Client
    private readonly topic : string;

    constructor() {
        // define your own constructor, e.g.
        this.topic = "<define_your_topic>";
        this.publishToIoTCore().then(r => console.log("Started workflow"));
```

```
    }

    private async publishToIoTCore() {
        try {
            const request: PublishToIoTCoreRequest = {
                topicName: this.topic,
                qos: QOS.AT_LEAST_ONCE, // you can change this depending on your use
 case
            }

            this.ipcClient = await getIpcClient();

            await this.ipcClient.publishToIoTCore(request);
        } catch (e) {
            // parse the error depending on your use cases
            throw e
        }
    }
}


export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}

// starting point
const publishToIoTCore = new PublishToIoTCore();
```

SubscribeToIoTCore

トピックまたはトピックフィルターに関する AWS IoT Core からの MQTT メッセージをサブスクライブします。コンポーネントがライフサイクルの終わりに達すると、AWS IoT Greengrass Core ソフトウェアはサブスクリプションを削除します。

このオペレーションはサブスクリプションオペレーションで、イベントメッセージのストリームをサブスクライブするというものです。このオペレーションを使用するには、イベントメッセージ、エラー、およびストリームクロージャを処理する関数を使用して、ストリームレスポンスハンドラーを定義します。(詳細については、IPC イベントストリームへのサブスクライブ を参照してください)。

イベントメッセージの種類: `IoTCoreMessage`

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`topicName` (Python: `topic_name`)

サブスクライブ先のトピック。MQTT トピックのワイルドカード (# および +) を使用して、複数のトピックにサブスクライブできます。

`qos`

使用する MQTT QoS。この列挙型 (QOS) には以下の値があります。

- `AT_MOST_ONCE` – QoS 0。MQTT メッセージが配信されるのは 1 回以下です。
- `AT_LEAST_ONCE` – QoS 1。MQTT メッセージが配信されるのは 1 回以上です。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`messages`

MQTT メッセージのストリーム。このオブジェクト (`IoTCoreMessage`) には、次の情報が含まれます。

`message`

MQTT メッセージ。このオブジェクト (`MQTTMessage`) には、次の情報が含まれます。

`topicName` (Python: `topic_name`)

メッセージが発行されたトピック。

`payload`

(オプション) BLOB としてのメッセージペイロード。

MQTT 5を使用する際、Greengrass nucleus の v2.10.0 以降で以下の機能が使用できます。MQTT 3.1.1 を使用している場合、これらの機能は無視されます。これらの機能を利用するために必要な AWS IoT デバイス SDK の最小バージョンは、次の表のとおりです。

SDK	最小バージョン
AWS IoT Device SDK for Python v2	v1.15.0
AWS IoT Device SDK for Java v2	v1.13.0
AWS IoT Device SDK for C++ v2	v1.24.0
AWS IoT Device SDK for JavaScript v2	v1.13.0

`payloadFormat`

(オプション) メッセージペイロードのフォーマット。`payloadFormat` を設定しない場合、タイプは BYTES とみなされます。この列挙型には以下の値があります。

- BYTES— ペイロードのコンテンツは、バイナリ BLOB です。
- UTF8— ペイロードのコンテンツは UTF8 の文字列です。

`retain`

(オプション) 発行時に MQTT 保持オプションを `true` に設定するか否かを示します。

`userProperties`

(オプション) 送信するアプリケーション固有の `UserProperty` オブジェクトのリストです。`UserProperty` オブジェクトは次のように定義されます。

```
UserProperty:
  key: string
  value: string
```

`messageExpiryIntervalSeconds`

(オプション) メッセージが期限切れとなり、サーバーによって削除されるまでの秒数です。この値を設定しない場合、メッセージに有効期限は設定されません。

`correlationData`

(オプション) リクエストに付加される情報で、リクエストとレスポンスの関連付けに使用できます。

`responseTopic`

(オプション) レスポンスメッセージに使用するトピックです。

`contentType`

(オプション)メッセージのコンテンツタイプのアプリケーション固有の識別子です。

例

以下の例では、カスタムコンポーネントコードでこのオペレーションを呼び出す方法を示します。

Java (IPC client V2)

Example 例:メッセージをサブスクライブする

```
package com.aws.greengrass.docs.samples.ipc;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientV2;
import software.amazon.awssdk.aws.greengrass.SubscribeToIoTCoreResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.QOS;
import software.amazon.awssdk.aws.greengrass.model.IoTCoreMessage;
import software.amazon.awssdk.aws.greengrass.model.SubscribeToIoTCoreRequest;
import software.amazon.awssdk.aws.greengrass.model.SubscribeToIoTCoreResponse;

import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.function.Consumer;
import java.util.function.Function;


public class SubscribeToIoTCore {

    public static void main(String[] args) {
        String topic = args[0];
```

```
        QOS qos = QOS.get(args[1]);

        Consumer<IoTCoreMessage> onStreamEvent = ioTCoreMessage ->
                System.out.printf("Received new message on topic %s: %s%n",
                        ioTCoreMessage.getMessage().getTopicName(),
                        new String(ioTCoreMessage.getMessage().getPayload(),
StandardCharsets.UTF_8));

        Optional<Function<Throwable, Boolean>> onStreamError =
                Optional.of(e -> {
                    System.err.println("Received a stream error.");
                    e.printStackTrace();
                    return false;
                });

        Optional<Runnable> onStreamClosed = Optional.of(() ->
                System.out.println("Subscribe to IoT Core stream closed."));

        try (GreengrassCoreIPCClientV2 ipcClientV2 =
GreengrassCoreIPCClientV2.builder().build()) {
            SubscribeToIoTCoreRequest request = new SubscribeToIoTCoreRequest()
                    .withTopicName(topic)
                    .withQos(qos);

            GreengrassCoreIPCClientV2.StreamingResponse<SubscribeToIoTCoreResponse,
SubscribeToIoTCoreResponseHandler>
                    streamingResponse = ipcClientV2.subscribeToIoTCore(request,
onStreamEvent, onStreamError, onStreamClosed);

            streamingResponse.getResponse();
            System.out.println("Successfully subscribed to topic: " + topic);

            // Keep the main thread alive, or the process will exit.
            while (true) {
                Thread.sleep(10000);
            }

            // To stop subscribing, close the stream.
            streamingResponse.getHandler().closeStream();
        } catch (InterruptedException e) {
            System.out.println("Subscribe interrupted.");
        } catch (Exception e) {
            System.err.println("Exception occurred.");
            e.printStackTrace();
```

```
            System.exit(1);
        }
    }
}
```

Python (IPC client V2)

Example 例: メッセージをサブスクライブする

> **Note**
>
> この例では、AWS IoT Device SDK for Python v2 のバージョン 1.5.4 以降を使用していることを前提としています。

```
import threading
import traceback

import awsiot.greengrasscoreipc.clientv2 as clientV2

topic = 'my/topic'
qos = '1'

def on_stream_event(event):
    try:
        topic_name = event.message.topic_name
        message = str(event.message.payload, 'utf-8')
        print(f'Received new message on topic {topic_name}:  {message}')
    except:
        traceback.print_exc()

def on_stream_error(error):
    # Return True to close stream, False to keep stream open.
    return True

def on_stream_closed():
    pass

ipc_client = clientV2.GreengrassCoreIPCClientV2()
resp, operation = ipc_client.subscribe_to_iot_core(
    topic_name=topic,
    qos=qos,
```

```
    on_stream_event=on_stream_event,
    on_stream_error=on_stream_error,
    on_stream_closed=on_stream_closed
)

# Keep the main thread alive, or the process will exit.
event = threading.Event()
event.wait()

# To stop subscribing, close the operation stream.
operation.close()
ipc_client.close()
```

Java (IPC client V1)

Example 例:メッセージをサブスクライブする

> **Note**
>
> この例は `IPCUtils` クラスを使用して、AWS IoT Greengrass Core IPC サービスへの接続を作成します。(詳細については、AWS IoT Greengrass Core IPC サービスに接続するを参照してください)。

```
package com.aws.greengrass.docs.samples.ipc;

import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.SubscribeToIoTCoreResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.*;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;
import software.amazon.awssdk.eventstreamrpc.StreamResponseHandler;

import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class SubscribeToIoTCore {
```

```
    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        String topic = args[0];
        QOS qos = QOS.get(args[1]);
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            StreamResponseHandler<IoTCoreMessage> streamResponseHandler =
                    new SubscriptionResponseHandler();
            SubscribeToIoTCoreResponseHandler responseHandler =
                    SubscribeToIoTCore.subscribeToIoTCore(ipcClient, topic, qos,
                            streamResponseHandler);
            CompletableFuture<SubscribeToIoTCoreResponse> futureResponse =
                    responseHandler.getResponse();
            try {
                futureResponse.get(TIMEOUT_SECONDS, TimeUnit.SECONDS);
                System.out.println("Successfully subscribed to topic: " + topic);
            } catch (TimeoutException e) {
                System.err.println("Timeout occurred while subscribing to topic: " +
topic);
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.println("Unauthorized error while subscribing to
topic: " + topic);
                } else {
                    throw e;
                }
            }

            // Keep the main thread alive, or the process will exit.
            try {
                while (true) {
                    Thread.sleep(10000);
                }
            } catch (InterruptedException e) {
                System.out.println("Subscribe interrupted.");
            }

            // To stop subscribing, close the stream.
            responseHandler.closeStream();
        } catch (InterruptedException e) {
            System.out.println("IPC interrupted.");
```

```
        } catch (ExecutionException e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static SubscribeToIoTCoreResponseHandler
subscribeToIoTCore(GreengrassCoreIPCClient greengrassCoreIPCClient, String topic,
QOS qos, StreamResponseHandler<IoTCoreMessage> streamResponseHandler) {
        SubscribeToIoTCoreRequest subscribeToIoTCoreRequest = new
SubscribeToIoTCoreRequest();
        subscribeToIoTCoreRequest.setTopicName(topic);
        subscribeToIoTCoreRequest.setQos(qos);
        return
greengrassCoreIPCClient.subscribeToIoTCore(subscribeToIoTCoreRequest,
                Optional.of(streamResponseHandler));
    }

    public static class SubscriptionResponseHandler implements
StreamResponseHandler<IoTCoreMessage> {

        @Override
        public void onStreamEvent(IoTCoreMessage ioTCoreMessage) {
            try {
                String topic = ioTCoreMessage.getMessage().getTopicName();
                String message = new
String(ioTCoreMessage.getMessage().getPayload(),
                        StandardCharsets.UTF_8);
                System.out.printf("Received new message on topic %s: %s%n", topic,
message);
            } catch (Exception e) {
                System.err.println("Exception occurred while processing subscription
response " +
                        "message.");
                e.printStackTrace();
            }
        }

        @Override
        public boolean onStreamError(Throwable error) {
            System.err.println("Received a stream error.");
            error.printStackTrace();
            return false;
```

```
        }

        @Override
        public void onStreamClosed() {
            System.out.println("Subscribe to IoT Core stream closed.");
        }
    }
}
```

Python (IPC client V1)

Example 例:メッセージをサブスクライブする

> **Note**
>
> この例では、AWS IoT Device SDK for Python v2 のバージョン 1.5.4 以降を使用していることを前提としています。

```
import time
import traceback

import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import (
    IoTCoreMessage,
    QOS,
    SubscribeToIoTCoreRequest
)

TIMEOUT = 10

ipc_client = awsiot.greengrasscoreipc.connect()

class StreamHandler(client.SubscribeToIoTCoreStreamHandler):
    def __init__(self):
        super().__init__()

    def on_stream_event(self, event: IoTCoreMessage) -> None:
        try:
            message = str(event.message.payload, "utf-8")
            topic_name = event.message.topic_name
            # Handle message.
```

```
        except:
            traceback.print_exc()

    def on_stream_error(self, error: Exception) -> bool:
        # Handle error.
        return True  # Return True to close stream, False to keep stream open.

    def on_stream_closed(self) -> None:
        # Handle close.
        pass


topic = "my/topic"
qos = QOS.AT_MOST_ONCE

request = SubscribeToIoTCoreRequest()
request.topic_name = topic
request.qos = qos
handler = StreamHandler()
operation = ipc_client.new_subscribe_to_iot_core(handler)
operation.activate(request)
future_response = operation.get_response()
future_response.result(TIMEOUT)

# Keep the main thread alive, or the process will exit.
while True:
    time.sleep(10)

# To stop subscribing, close the operation stream.
operation.close()
```

C++

Example 例:メッセージをサブスクライブする

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IoTCoreResponseHandler : public SubscribeToIoTCoreStreamHandler {
```

```
    public:
        virtual ~IoTCoreResponseHandler() {}

    private:
        void OnStreamEvent(IoTCoreMessage *response) override {
            auto message = response->GetMessage();
            if (message.has_value() && message.value().GetPayload().has_value()) {
                auto messageBytes = message.value().GetPayload().value();
                std::string messageString(messageBytes.begin(), messageBytes.end());
                std::string topicName =
 message.value().GetTopicName().value().c_str();
                // Handle message.
            }
        }

        bool OnStreamError(OperationError *error) override {
            // Handle error.
            return false; // Return true to close stream, false to keep stream open.
        }

        void OnStreamClosed() override {
            // Handle close.
        }
};

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }

    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }

    bool OnErrorCallback(RpcError error) override {
        // Handle IPC service connection error.
        return true;
    }
};

int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
```

```
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    String topic("my/topic");
    QOS qos = QOS_AT_MOST_ONCE;
    int timeout = 10;

    SubscribeToIoTCoreRequest request;
    request.SetTopicName(topic);
    request.SetQos(qos);
    auto streamHandler = MakeShared<IoTCoreResponseHandler>(DefaultAllocator());
    auto operation = ipcClient.NewSubscribeToIoTCore(streamHandler);
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (!response) {
        // Handle error.
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            (void)error;
            // Handle operation error.
        } else {
            // Handle RPC error.
        }
        exit(-1);
    }
```

```
    // Keep the main thread alive, or the process will exit.
    while (true) {
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    operation->Close();
    return 0;
}
```

JavaScript

Example 例:メッセージをサブスクライブする

```
import * as greengrasscoreipc from "aws-iot-device-sdk-v2/dist/greengrasscoreipc";
import {IoTCoreMessage, QOS, SubscribeToIoTCoreRequest} from "aws-iot-device-sdk-v2/
dist/greengrasscoreipc/model";
import {RpcError} from "aws-iot-device-sdk-v2/dist/eventstream_rpc";

class SubscribeToIoTCore {
    private ipcClient : greengrasscoreipc.Client
    private readonly topic : string;

    constructor() {
        // define your own constructor, e.g.
        this.topic = "<define_your_topic>";
        this.subscribeToIoTCore().then(r => console.log("Started workflow"));
    }

    private async subscribeToIoTCore() {
        try {
            const request: SubscribeToIoTCoreRequest = {
                topicName: this.topic,
                qos: QOS.AT_LEAST_ONCE, // you can change this depending on your use
 case
            }

            this.ipcClient = await getIpcClient();

            const streamingOperation = this.ipcClient.subscribeToIoTCore(request);

            streamingOperation.on('message', (message: IoTCoreMessage) => {
                // parse the message depending on your use cases, e.g.
```

```
                if (message.message && message.message.payload) {
                    const receivedMessage = message.message.payload.toString();
                }
            });

            streamingOperation.on('streamError', (error : RpcError) => {
                // define your own error handling logic
            });

            streamingOperation.on('ended', () => {
                // define your own logic
            });

            await streamingOperation.activate();

            // Keep the main thread alive, or the process will exit.
            await new Promise((resolve) => setTimeout(resolve, 10000))
        } catch (e) {
            // parse the error depending on your use cases
            throw e
        }
    }
}

export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}

// starting point
const subscribeToIoTCore = new SubscribeToIoTCore();
```

例

コンポーネントの AWS IoT Core MQTT IPC サービスの使用方法については、以下の例を参照してください。

AWS IoT Core MQTT パブリッシャー (C++) の例

以下の recipe の例は、コンポーネントをすべてのトピックに発行できるようにします。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.IoTCorePublisherCpp",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes MQTT messages to IoT Core.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.IoTCorePublisherCpp:mqttproxy:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": [
              "aws.greengrass#PublishToIoTCore"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "run": "{artifacts:path}/greengrassv2_iotcore_publisher"
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.IoTCorePublisherCpp/1.0.0/greengrassv2_iotcore_publisher",
```

```
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.IoTCorePublisherCpp
ComponentVersion: 1.0.0
ComponentDescription: A component that publishes MQTT messages to IoT Core.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.mqttproxy:
        com.example.IoTCorePublisherCpp:mqttproxy:1:
          policyDescription: Allows access to publish to all topics.
          operations:
            - aws.greengrass#PublishToIoTCore
          resources:
            - "*"
Manifests:
  - Lifecycle:
      run: "{artifacts:path}/greengrassv2_iotcore_publisher"
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.IoTCorePublisherCpp/1.0.0/greengrassv2_iotcore_publisher
        Permission:
          Execute: OWNER
```

以下の C++ アプリケーションの例は、AWS IoT Core MQTT IPC サービスを使用して AWS IoT Core にメッセージを発行する方法を示します。

```
#include <iostream>

#include <aws/crt/Api.h>
```

```
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
    }

    void OnDisconnectCallback(RpcError error) override {
        std::cout << "OnDisconnectCallback: " << error.StatusToString() << std::endl;
        exit(-1);
    }

    bool OnErrorCallback(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
};

int main() {
    String message("Hello from the Greengrass IPC MQTT publisher (C++).");
    String topic("test/topic/cpp");
    QOS qos = QOS_AT_LEAST_ONCE;
    int timeout = 10;

    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
 connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    while (true) {
        PublishToIoTCoreRequest request;
        Vector<uint8_t> messageData({message.begin(), message.end()});
        request.SetTopicName(topic);
```

```
        request.SetPayload(messageData);
        request.SetQos(qos);

        auto operation = ipcClient.NewPublishToIoTCore();
        auto activate = operation->Activate(request, nullptr);
        activate.wait();

        auto responseFuture = operation->GetResult();
        if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
 std::future_status::timeout) {
            std::cerr << "Operation timed out while waiting for response from
 Greengrass Core." << std::endl;
            exit(-1);
        }

        auto response = responseFuture.get();
        if (response) {
            std::cout << "Successfully published to topic: " << topic << std::endl;
        } else {
            // An error occurred.
            std::cout << "Failed to publish to topic: " << topic << std::endl;
            auto errorType = response.GetResultType();
            if (errorType == OPERATION_ERROR) {
                auto *error = response.GetOperationError();
                std::cout << "Operation error: " << error->GetMessage().value() <<
 std::endl;
            } else {
                std::cout << "RPC error: " << response.GetRpcError() << std::endl;
            }
            exit(-1);
        }

        std::this_thread::sleep_for(std::chrono::seconds(5));
    }

    return 0;
}
```

AWS IoT Core MQTT サブスクライバー (C++) の例

以下の recipe の例は、コンポーネントをすべてのトピックをサブスクライブできるようにします。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.IoTCoreSubscriberCpp",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to MQTT messages from IoT
 Core.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.IoTCoreSubscriberCpp:mqttproxy:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": [
              "aws.greengrass#SubscribeToIoTCore"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "run": "{artifacts:path}/greengrassv2_iotcore_subscriber"
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.IoTCoreSubscriberCpp/1.0.0/greengrassv2_iotcore_subscriber",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.IoTCoreSubscriberCpp
ComponentVersion: 1.0.0
ComponentDescription: A component that subscribes to MQTT messages from IoT Core.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.mqttproxy:
        com.example.IoTCoreSubscriberCpp:mqttproxy:1:
          policyDescription: Allows access to subscribe to all topics.
          operations:
            - aws.greengrass#SubscribeToIoTCore
          resources:
            - "*"
Manifests:
  - Lifecycle:
      run: "{artifacts:path}/greengrassv2_iotcore_subscriber"
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.IoTCoreSubscriberCpp/1.0.0/greengrassv2_iotcore_subscriber
        Permission:
          Execute: OWNER
```

以下の C++ アプリケーションの例は、AWS IoT Core MQTT IPC サービスを使用して AWS IoT Core からメッセージをサブスクライブする方法を示します。

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IoTCoreResponseHandler : public SubscribeToIoTCoreStreamHandler {

    public:
        virtual ~IoTCoreResponseHandler() {}
```

```
    private:

        void OnStreamEvent(IoTCoreMessage *response) override {
            auto message = response->GetMessage();
            if (message.has_value() && message.value().GetPayload().has_value()) {
                auto messageBytes = message.value().GetPayload().value();
                std::string messageString(messageBytes.begin(), messageBytes.end());
                std::string messageTopic =
 message.value().GetTopicName().value().c_str();
                std::cout << "Received new message on topic: " << messageTopic <<
 std::endl;
                std::cout << "Message: " << messageString << std::endl;
            }
        }

        bool OnStreamError(OperationError *error) override {
            std::cout << "Received an operation error: ";
            if (error->GetMessage().has_value()) {
                std::cout << error->GetMessage().value();
            }
            std::cout << std::endl;
            return false; // Return true to close stream, false to keep stream open.
        }

        void OnStreamClosed() override {
            std::cout << "Subscribe to IoT Core stream closed." << std::endl;
        }
};

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
    }

    void OnDisconnectCallback(RpcError error) override {
        std::cout << "OnDisconnectCallback: " << error.StatusToString() << std::endl;
        exit(-1);
    }

    bool OnErrorCallback(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
```

```
};

int main() {
    String topic("test/topic/cpp");
    QOS qos = QOS_AT_LEAST_ONCE;
    int timeout = 10;

    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
 connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    SubscribeToIoTCoreRequest request;
    request.SetTopicName(topic);
    request.SetQos(qos);
    auto streamHandler = MakeShared<IoTCoreResponseHandler>(DefaultAllocator());
    auto operation = ipcClient.NewSubscribeToIoTCore(streamHandler);
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
 std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
 Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (response) {
        std::cout << "Successfully subscribed to topic: " << topic << std::endl;
    } else {
        // An error occurred.
        std::cout << "Failed to subscribe to topic: " << topic << std::endl;
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
```

```
            auto *error = response.GetOperationError();
            std::cout << "Operation error: " << error->GetMessage().value() <<
 std::endl;
        } else {
            std::cout << "RPC error: " << response.GetRpcError() << std::endl;
        }
        exit(-1);
    }

    // Keep the main thread alive, or the process will exit.
    while (true) {
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    operation->Close();
    return 0;
}
```

コンポーネントライフサイクルの操作

コンポーネントライフサイクルの IPC サービスを使用して、次を行います。

- コアデバイスのコンポーネント状態の更新。
- コンポーネント状態の更新のサブスクライブ。
- デプロイ中に nucleus によりコンポーネントが停止され、更新が適用されることの防止。
- コンポーネントのプロセスの一時停止と再開。

トピック

- 最小 SDK バージョン
- 認証
- UpdateState
- SubscribeToComponentUpdates
- DeferComponentUpdate
- PauseComponent
- ResumeComponent

最小 SDK バージョン

次の表に、AWS IoT Device SDK の最小バージョンを示します。コンポーネントのライフサイクルを操作する際は、これを使用する必要があります。

SDK	最小バージョン	
AWS IoT Device SDK for Java v2	v1.2.10	
AWS IoT Device SDK for Python v2	v1.5.3	
AWS IoT Device SDK for C++ v2	v1.17.0	
AWS IoT Device SDK JavaScript v2 用	v1.12.0	

認証

カスタムコンポーネントから他のコンポーネントを一時停止または再開するには、コンポーネントが他のコンポーネントを管理できるようにする承認ポリシーを定義する必要があります。承認ポリシーの定義については、「コンポーネントに IPC オペレーションの実行を許可する」を参照してください。

コンポーネントライフサイクル管理の承認ポリシーには、次のプロパティがあります。

IPC サービス識別子: `aws.greengrass.ipc.lifecycle`

操作	説明	リソース
`aws.greengrass#PauseComponent`	コンポーネントが、指定したコンポーネントを一時停止できるようにします。	すべてのコンポーネントへのアクセスを許可するコンポーネント名または *。

操作	説明	リソース
aws.greengrass#ResumeComponent	コンポーネントが、指定したコンポーネントを再開できるようにします。	すべてのコンポーネントへのアクセスを許可するコンポーネント名または *。
*	コンポーネントが、指定したコンポーネントを一時停止および再開できるようにします。	すべてのコンポーネントへのアクセスを許可するコンポーネント名または *。

承認ポリシーの例

次の承認ポリシーの例を参照して、コンポーネントの承認ポリシーの設定に役立てることができます。

Example 承認ポリシーの例

次の承認ポリシーの例では、コンポーネントが、すべてのコンポーネントを一時停止および再開できるようにします。

```
{
  "accessControl": {
    "aws.greengrass.ipc.lifecycle": {
      "com.example.MyLocalLifecycleComponent:lifecycle:1": {
        "policyDescription": "Allows access to pause/resume all components.",
        "operations": [
          "aws.greengrass#PauseComponent",
          "aws.greengrass#ResumeComponent"
        ],
        "resources": [
          "*"
        ]
      }
    }
  }
}
```

UpdateState

コアデバイスのコンポーネントの状態を更新します。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`state`

設定する状態。この列挙型 (`LifecycleState`) には以下の値があります。

- `RUNNING`
- `ERRORED`

レスポンス

このオペレーションはレスポンスで一切の情報を提供しません。

SubscribeToComponentUpdates

AWS IoT Greengrass Core ソフトウェアがコンポーネントを更新する前に通知を受け取るようにサブスクライブします。通知では、更新の一部として nucleus を再起動するかどうかを指定します。

nucleus は、デプロイのコンポーネント更新ポリシーがコンポーネントに通知するように指定されている場合のみ、更新通知を送信します。デフォルトの動作では、コンポーネントに通知を行います。詳細については、デプロイの作成「」と、 CreateDeploymentオペレーションを呼び出すときに提供できる DeploymentComponentUpdatePolicy オブジェクトを参照してください。

> **Important**
> 更新前にローカルデプロイがコンポーネントに通知を行うことばありません。

このオペレーションはサブスクリプションオペレーションで、イベントメッセージのストリームをサブスクライブするというものです。このオペレーションを使用するには、イベントメッセージ、エラー、およびストリームクロージャを処理する関数を使用して、ストリームレスポンスハンドラーを定義します。(詳細については、IPC イベントストリームへのサブスクライブ を参照してください)。

イベントメッセージの種類: `ComponentUpdatePolicyEvents`

> **Tip**
>
> チュートリアルに従って、条件付きでコンポーネントの更新を延期するコンポーネントを開発する方法を説明します。(詳細については、チュートリアル: コンポーネントの更新を延期する Greengrass コンポーネントを開発する を参照してください)。

リクエスト

このオペレーションのリクエストはパラメータを持ちません。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`messages`

通知メッセージのストリーム。このオブジェクト (`ComponentUpdatePolicyEvents`) には、次の情報が含まれます。

`preUpdateEvent` (Python: `pre_update_event`)

(オプション) nucleus がコンポーネントの更新を希望することを示すイベント。DeferComponentUpdate オペレーションで応答して更新を承認するか、コンポーネントを再起動する準備が整うまで延期するかを選択できます。このオブジェクト (`PreComponentUpdateEvent`) には、次の情報が含まれます。

`deploymentId` (Python: `deployment_id`)

コンポーネントを更新する AWS IoT Greengrass デプロイの ID。

`isGgcRestarting` (Python: `is_ggc_restarting`)

更新を適用するために nucleus を再起動する必要があるかどうか。

`postUpdateEvent` (Python: `post_update_event`)

(オプション) nucleus がコンポーネントを更新したことを示すイベント。このオブジェクト (`PostComponentUpdateEvent`) には、次の情報が含まれます。

`deploymentId` (Python: `deployment_id`)

コンポーネントを更新した AWS IoT Greengrass デプロイの ID。

> **Note**
>
> この機能を使用するには、Greengrass nucleus コンポーネントの v2.7.0 以降が必要です。

DeferComponentUpdate

SubscribeToComponentUpdates で検出したコンポーネントの更新を承認または延期します。コンポーネントが更新を進める準備ができているかどうか、nucleus が再度チェックするまで待機する時間を指定します。このオペレーションを使用して、コンポーネントが更新の準備ができていることを nucleus に通知することもできます。

コンポーネントがコンポーネント更新通知に応答しない場合、nucleus はデプロイのコンポーネント更新ポリシーで指定した時間だけ待機します。これがタイムアウトした後、nucleus はデプロイを続行します。デフォルトのコンポーネント更新のタイムアウトは 60 秒です。詳細については、デプロイの作成「」と、 CreateDeploymentオペレーションを呼び出すときに指定できる DeploymentComponentUpdatePolicy オブジェクトを参照してください。

> **Tip**
>
> チュートリアルに従って、条件付きでコンポーネントの更新を延期するコンポーネントを開発する方法を説明します。(詳細については、チュートリアル: コンポーネントの更新を延期する Greengrass コンポーネントを開発する を参照してください)。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`deploymentId` (Python: `deployment_id`)

延期する AWS IoT Greengrass デプロイのID。

`message`

(オプション) 更新を延期するコンポーネントの名前。

デフォルトでは、リクエストを実行するコンポーネントの名前になっています。

`recheckAfterMs` (Python: `recheck_after_ms`)

更新を延期する時間 (ミリ秒)。nucleus はこの時間だけ待機してから、SubscribeToComponentUpdates で発見できる別の `PreComponentUpdateEvent` を送信します。

`0` を指定すると更新が承認されます。これにより、コンポーネントが更新の準備ができていることが nucleus に通知されます。

デフォルトは 0 ミリ秒で、これは更新を承認することを意味します。

レスポンス

このオペレーションはレスポンスで一切の情報を提供しません。

PauseComponent

この機能は、Greengrass nucleus コンポーネントの v2.4.0 以降に利用できます。AWS IoT Greengrass は、現在 Windows コアデバイスにこの機能をサポートしていません。

コアデバイスのコンポーネントのプロセスを一時停止します。コンポーネントを再開するには、ResumeComponentオペレーションを使用します。

一時停止できるのはジェネリックコンポーネントのみです。他の種類のコンポーネントを一時停止しようとすると、オペレーションは `InvalidRequestError` をスローします。

> **Note**
>
> この操作は、Docker コンテナなどのコンテナ化されたプロセスは一時停止できません。Docker コンテナの一時停止および再開を行うには、docker pause および docker unpause コマンドを使用できます。

このオペレーションは、コンポーネントの依存関係や、一時停止するコンポーネントに依存するコンポーネントに対しては、一時停止を行いません。別のコンポーネントの依存関係であるコンポーネントを一時停止するときは、この動作を検討してください。依存するコンポーネントは、依存関係が一時停止されたときに問題が発生する可能性があるためです。

デプロイ経由などで一時停止したコンポーネントを再起動またはシャットダウンすると、Greengrass nucleus はコンポーネントを再開し、シャットダウンライフサイクルを実行します。コンポーネントの再起動についての詳細は、「RestartComponent」を参照してください。

> **Important**
>
> このオペレーションを使用するには、このオペレーションを使用する権限を付与する承認ポリシーを定義する必要があります。(詳細については、認証 を参照してください)。

最小 SDK バージョン

以下の表に AWS IoT Device SDK の最小バージョンを示します。コンポーネントを一時停止および再開する際は、これを使用する必要があります。

SDK	最小バージョン
AWS IoT Device SDK for Java v2	v1.4.3
AWS IoT Device SDK for Python v2	v1.6.2
AWS IoT Device SDK for C++ v2	v1.13.1
AWS IoT Device SDK JavaScript v2 用	v1.12.0

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`componentName` (Python: `component_name`)

一時停止するコンポーネントの名前。ジェネリックコンポーネントである必要があります。(詳細については、コンポーネントタイプ を参照してください)。

レスポンス

このオペレーションはレスポンスで一切の情報を提供しません。

ResumeComponent

この機能は、Greengrass nucleus コンポーネントの v2.4.0 以降に利用できます。AWS IoT Greengrass は、現在 Windows コアデバイスにこの機能をサポートしていません。

コアデバイスのコンポーネントのプロセスを再開します。コンポーネントを一時停止するには、PauseComponentオペレーションを使用します。

再開できるのは一時停止しているコンポーネントのみです。一時停止していないコンポーネントを再開しようとすると、このオペレーションは `InvalidRequestError` をスローします。

> **Important**
>
> このオペレーションを使用するには、そうするための権限を付与する承認ポリシーを定義する必要があります。(詳細については、認証 を参照してください)。

最小 SDK バージョン

以下の表に AWS IoT Device SDK の最小バージョンを示します。コンポーネントを一時停止および再開する際は、これを使用する必要があります。

SDK	最小バージョン	
AWS IoT Device SDK for Java v2	v1.4.3	
AWS IoT Device SDK for Python v2	v1.6.2	
AWS IoT Device SDK for C++ v2	v1.13.1	
AWS IoT Device SDK JavaScript v2 用	v1.12.0	

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`componentName` (Python: `component_name`)

再開するコンポーネントの名前。

レスポンス

このオペレーションはレスポンスで一切の情報を提供しません。

コンポーネント設定とやり取り

コンポーネント設定 IPC サービスでは以下のことが行えます。

- コンポーネント設定パラメータを取得および設定します。
- コンポーネント設定の更新をサブスクライブします。
- コンポーネント設定の更新を、nucleus が適用する前に検証します。

トピック

最小 SDK バージョン

以下の表に AWS IoT Device SDK の最小バージョンを示します。コンポーネント設定とやり取りする際は、これを使用する必要があります。

SDK	最小バージョン	
AWS IoT Device SDK for Java v2	v1.2.10	
AWS IoT Device SDK for Python v2	v1.5.3	
AWS IoT Device SDK for C++ v2	v1.17.0	
AWS IoT Device SDK JavaScript v2 用	v1.12.0	

GetConfiguration

コアデバイス上のコンポーネントの設定値を取得します。設定値を取得するためのキーパスを指定します。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`componentName` (Python: `component_name`)

(オプション) コンポーネントの名前。

デフォルトでは、リクエストを実行するコンポーネントの名前になっています。

`keyPath` (Python: `key_path`)

設定値へのキーパス。各エントリが設定オブジェクトの単一レベルのキーとなるリストを指定します。たとえば、以下の設定で `port` の値を取得するには、`["mqtt", "port"]` を指定します。

```
{
  "mqtt": {
    "port": 443
  }
```

```
}
```

コンポーネントの完全な設定を取得するには、空のリストを指定します。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

componentName (Python: component_name)

コンポーネントの名前。

value

オブジェクトとしてリクエストされた設定。

UpdateConfiguration

コアデバイス上のこのコンポーネントの設定値を更新します。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

keyPath (Python: key_path)

(オプション) 更新するコンテナノード (オブジェクト) へのキーパス。各エントリが設定オブジェクトの単一レベルのキーとなるリストを指定します。たとえば、キーパス ["mqtt"] とマージ値 { "port": 443 } を指定して、以下の設定に port の値を設定します。

```
{
  "mqtt": {
    "port": 443
  }
}
```

キーパスは、設定内のコンテナノード (オブジェクト) を指定する必要があります。コンポーネントの設定にノードが存在しない場合、このオペレーションによってノードが作成され、valueToMerge のオブジェクトにその値が設定されます。

設定オブジェクトのルートに対するデフォルトです。

`timestamp`

現在の UNIX エポック時間 (ミリ秒)。このオペレーションにおけるキーの同時更新の解決には、このタイムスタンプが使用されます。コンポーネント設定のキーのタイムスタンプがリクエストのタイムスタンプよりも大きい場合、リクエストは失敗します。

`valueToMerge` (Python: `value_to_merge`)

keyPath で指定した場所でマージする設定オブジェクト。(詳細については、コンポーネント設定の更新 を参照してください)。

レスポンス

このオペレーションはレスポンスで一切の情報を提供しません。

SubscribeToConfigurationUpdate

サブスクライブして、コンポーネントの設定更新時に通知を受け取るようにします。キーをサブスクライブすると、そのキーの子が更新されたとき通知を受けます。

このオペレーションはサブスクリプションオペレーションで、イベントメッセージのストリームをサブスクライブするというものです。このオペレーションを使用するには、イベントメッセージ、エラー、およびストリームクロージャを処理する関数を使用して、ストリームレスポンスハンドラーを定義します。(詳細については、IPC イベントストリームへのサブスクライブ を参照してください)。

イベントメッセージの種類: `ConfigurationUpdateEvents`

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`componentName` (Python: `component_name`)

(オプション) コンポーネントの名前。

デフォルトでは、リクエストを実行するコンポーネントの名前になっています。

`keyPath` (Python: `key_path`)

サブスクライブする設定値のキーパス。各エントリが設定オブジェクトの単一レベルのキーとなるリストを指定します。たとえば、以下の設定で `port` の値を取得するには、`["mqtt", "port"]` を指定します。

```
{
  "mqtt": {
    "port": 443
  }
}
```

コンポーネント設定のすべての値の更新をサブスクライブするには、空のリストを指定します。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`messages`

通知メッセージのストリーム。このオブジェクト (`ConfigurationUpdateEvents`) には、次の情報が含まれます。

`configurationUpdateEvent` (Python: `configuration_update_event`)

設定更新イベント。このオブジェクト (`ConfigurationUpdateEvent`) には、次の情報が含まれます。

`componentName` (Python: `component_name`)

コンポーネントの名前。

`keyPath` (Python: `key_path`)

更新された設定値へのキーパス。

SubscribeToValidateConfigurationUpdates

サブスクライブして、このコンポーネントの設定更新より前に通知を受け取るようにします。これにより、コンポーネントは各自の設定に対し更新を検証できます。SendConfigurationValidityReport オペレーションを使用して、設定が有効かどうかを nucleus に伝えます。

> **Important**
>
> ローカルデプロイは、コンポーネントに更新を通知しません。

このオペレーションはサブスクリプションオペレーションで、イベントメッセージのストリームをサブスクライブするというものです。このオペレーションを使用するには、イベントメッセージ、エラー、およびストリームクロージャを処理する関数を使用して、ストリームレスポンスハンドラーを定義します。(詳細については、IPC イベントストリームへのサブスクライブ を参照してください)。

イベントメッセージの種類: `ValidateConfigurationUpdateEvents`

リクエスト

このオペレーションのリクエストはパラメータを持ちません。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`messages`

通知メッセージのストリーム。このオブジェクト (`ValidateConfigurationUpdateEvents`) には、次の情報が含まれます。

`validateConfigurationUpdateEvent` (Python: `validate_configuration_update_event`)

設定更新イベント。このオブジェクト (`ValidateConfigurationUpdateEvent`) には、次の情報が含まれます。

`deploymentId` (Python: `deployment_id`)

コンポーネントを更新する AWS IoT Greengrass デプロイの ID。

`configuration`

新しい設定を含むオブジェクト。

SendConfigurationValidityReport

このコンポーネントの設定更新が有効かどうかをnucleus に伝えます。新しい設定が有効でないことを nucleus に伝えると、デプロイは失敗します。SubscribeToValidateConfigurationUpdates オペレーションを使用して、設定更新の検証をサブスクライブします。

コンポーネントが設定更新の検証の通知に応答しない場合、nucleus はデプロイの設定検証ポリシーで指定した時間だけ待機します。これがタイムアウトした後、nucleus はデプロイを続行します。デフォルトのコンポーネント検証のタイムアウトは 20 秒です。詳細について

は、デプロイの作成「」と、 CreateDeploymentオペレーションを呼び出すときに指定できる DeploymentConfigurationValidationPolicy オブジェクトを参照してください。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`configurationValidityReport` (Python: `configuration_validity_report`)

設定更新が有効かどうかを nucleus に伝えるレポート。このオブジェクト (`ConfigurationValidityReport`) には、次の情報が含まれます。

`status`

有効性のステータス。この列挙型 (`ConfigurationValidityStatus`) には以下の値があります。

- `ACCEPTED` - 設定が有効で、nucleus はこのコンポーネントに適用できます。
- `REJECTED` - 設定が有効ではなく、デプロイが失敗します。

`deploymentId` (Python: `deployment_id`)

設定更新をリクエストした AWS IoT Greengrass デプロイの ID。

`message`

(オプション) 設定が有効でないことの理由を報告するメッセージ。

レスポンス

このオペレーションはレスポンスで一切の情報を提供しません。

シークレット値を取得する

シークレットマネージャー IPC サービスを使用して、コアデバイスのシークレットからシークレット値を取得します。シークレットマネージャーコンポーネントを使用して、暗号化されたシークレットをコアデバイスにデプロイします。次に、IPC オペレーションを使用してシークレットを復号化し、カスタムコンポーネントでその値を使用できます。

トピック

- 最小 SDK バージョン
- 認証

- GetSecretValue
- 例

最小 SDK バージョン

次の表に AWS IoT Device SDK の最小バージョンを一覧表示します。これは、コアデバイスのシークレットからシークレット値を取得するために使用する必要があります。

SDK	最小バージョン	
AWS IoT Device SDK for Java v2	v1.2.10	
AWS IoT Device SDK for Python v2	v1.5.3	
AWS IoT Device SDK for C++ v2	v1.17.0	
AWS IoT Device SDK JavaScript v2 用	v1.12.0	

認証

カスタムコンポーネントでシークレットマネージャーを使用するには、コアデバイスに保存するシークレットの値をコンポーネントが取得できるようにする承認ポリシーを定義する必要があります。承認ポリシーの定義については、「コンポーネントに IPC オペレーションの実行を許可する」を参照してください。

シークレットマネージャーの承認ポリシーには以下のプロパティがあります。

IPC サービス識別子: `aws.greengrass.SecretManager`

操作	説明	リソース
`aws.greengrass#GetSecretValue` または *	コンポーネントがコアデバイスで暗号化されたシークレッ	Secrets Manager のシークレット ARN、または * ですべ

操作	説明	リソース
	トの値を取得できるようにします。	てのシークレットへのアクセスを許可します。

承認ポリシーの例

次の承認ポリシーの例を参照して、コンポーネントの承認ポリシーの設定に役立てることができます。

Example 承認ポリシーの例

以下の承認ポリシーの例は、コンポーネントがコアデバイスのすべてのシークレット値を取得できるようにします。

> **Note**
>
> 実稼働環境では、コンポーネントが使用するシークレットのみ取得するように、承認ポリシーの範囲を小さくすることをお勧めします。コンポーネントをデプロイするとき、* ワイルドカードをシークレット ARN のリストに変更できます。

```
{
  "accessControl": {
    "aws.greengrass.SecretManager": {
      "com.example.MySecretComponent:secrets:1": {
        "policyDescription": "Allows access to a secret.",
        "operations": [
          "aws.greengrass#GetSecretValue"
        ],
        "resources": [
          "*"
        ]
      }
    }
  }
}
```

GetSecretValue

コアデバイスに保存するシークレットの値を取得します。

このオペレーションは Secrets Manager のオペレーションと似ていますが、AWS クラウド のシークレット値を取得するために使用できます。詳細については、AWS Secrets Manager API リファレンスの「GetSecretValue」を参照してください。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`secretId` (Python: `secret_id`)

取得するシークレットの名前。シークレットの Amazon リソースネーム (ARN) またはフレンドリ名を指定します。

`versionId` (Python: `version_id`)

(オプション) 取得するバージョンの ID。

`versionId` または `versionStage` のどちらかを指定できます。

`versionId` または `versionStage` を指定しない場合、このオペレーションはデフォルトで `AWSCURRENT` ラベルのバージョンになります。

`versionStage` (Python: `version_stage`)

(オプション) 取得するバージョンのステージングラベル。

`versionId` または `versionStage` のどちらかを指定できます。

`versionId` または `versionStage` を指定しない場合、このオペレーションはデフォルトで `AWSCURRENT` ラベルのバージョンになります。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`secretId` (Python: `secret_id`)

シークレットのID。

`versionId` (Python: `version_id`)

このバージョンのシークレットの ID。

`versionStage` (Python: `version_stage`)

このシークレットのバージョンには、ステージングラベルのリストがアタッチされています。

`secretValue` (Python: `secret_value`)

このバージョンのシークレットの値。このオブジェクト (`SecretValue`) には、次の情報が含まれます。

`secretString` (Python: `secret_string`)

Secrets Manager に文字列として提供した、保護されたシークレット情報の復号化された部分。

`secretBinary` (Python: `secret_binary`)

(オプション) Secrets Manager にバイト配列のバイナリデータとして提供した、保護されたシークレット情報の復号化された部分。このプロパティには、base64 エンコードされた文字列としてのバイナリデータが含まれています。

Secrets Manager コンソールでシークレットを作成した場合、このプロパティは使用されません。

例

以下の例では、カスタムコンポーネントコードでこのオペレーションを呼び出す方法を示します。

Java (IPC client V1)

Example 例: シークレット値の取得

> **Note**
>
> この例は `IPCUtils` クラスを使用して、AWS IoT Greengrass Core IPC サービスへの接続を作成します。詳細については、「AWS IoT Greengrass Core IPC サービスに接続する」を参照してください。

```
package com.aws.greengrass.docs.samples.ipc;
```

```
import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.GetSecretValueResponseHandler;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.model.GetSecretValueRequest;
import software.amazon.awssdk.aws.greengrass.model.GetSecretValueResponse;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;

import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class GetSecretValue {

    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        String secretArn = args[0];
        String versionStage = args[1];
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            GetSecretValueResponseHandler responseHandler =
                    GetSecretValue.getSecretValue(ipcClient, secretArn,
 versionStage);
            CompletableFuture<GetSecretValueResponse> futureResponse =
                    responseHandler.getResponse();
            try {
                GetSecretValueResponse response =
 futureResponse.get(TIMEOUT_SECONDS, TimeUnit.SECONDS);
                response.getSecretValue().postFromJson();
                String secretString = response.getSecretValue().getSecretString();
                System.out.println("Successfully retrieved secret value: " +
 secretString);
            } catch (TimeoutException e) {
                System.err.println("Timeout occurred while retrieving secret: " +
 secretArn);
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.println("Unauthorized error while retrieving secret:
 " + secretArn);
```

```
                } else {
                    throw e;
                }
            }
        } catch (InterruptedException e) {
            System.out.println("IPC interrupted.");
        } catch (ExecutionException e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static GetSecretValueResponseHandler
 getSecretValue(GreengrassCoreIPCClient greengrassCoreIPCClient, String secretArn,
 String versionStage) {
        GetSecretValueRequest getSecretValueRequest = new GetSecretValueRequest();
        getSecretValueRequest.setSecretId(secretArn);
        getSecretValueRequest.setVersionStage(versionStage);
        return greengrassCoreIPCClient.getSecretValue(getSecretValueRequest,
 Optional.empty());
    }
}
```

Python (IPC client V1)

Example 例: シークレット値の取得

> **Note**
>
> この例では、AWS IoT Device SDK for Python v2 のバージョン 1.5.4 以降を使用していることを前提としています。

```
import json

import awsiot.greengrasscoreipc
from awsiot.greengrasscoreipc.model import (
    GetSecretValueRequest,
    GetSecretValueResponse,
    UnauthorizedError
```

```
)

secret_id = 'arn:aws:secretsmanager:us-
west-2:123456789012:secret:MyGreengrassSecret-abcdef'
TIMEOUT = 10

ipc_client = awsiot.greengrasscoreipc.connect()

request = GetSecretValueRequest()
request.secret_id = secret_id
request.version_stage = 'AWSCURRENT'
operation = ipc_client.new_get_secret_value()
operation.activate(request)
future_response = operation.get_response()
response = future_response.result(TIMEOUT)
secret_json = json.loads(response.secret_value.secret_string)
# Handle secret value.
```

JavaScript

Example 例: シークレット値の取得

```
import {
    GetSecretValueRequest,
} from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc/model';
import * as greengrasscoreipc from "aws-iot-device-sdk-v2/dist/greengrasscoreipc";

class GetSecretValue {
    private readonly secretId : string;
    private readonly versionStage : string;
    private ipcClient : greengrasscoreipc.Client

    constructor() {
        this.secretId = "<define_your_own_secretId>"
        this.versionStage = "<define_your_own_versionStage>"

        this.getSecretValue().then(r => console.log("Started workflow"));
    }

    private async getSecretValue() {
        try {
            this.ipcClient = await getIpcClient();
```

```
            const getSecretValueRequest : GetSecretValueRequest = {
                secretId: this.secretId,
                versionStage: this.versionStage,
            };

            const result = await
 this.ipcClient.getSecretValue(getSecretValueRequest);
            const secretString = result.secretValue.secretString;
            console.log("Successfully retrieved secret value: " + secretString)
        } catch (e) {
            // parse the error depending on your use cases
            throw e
        }
    }
}

export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}

const getSecretValue = new GetSecretValue();
```

例

コンポーネントのシークレットマネージャー IPC サービスの使用方法については、以下の例を参照してください。

例: シークレットを出力 (Python、IPC クライアント V1)

このコンポーネントの例は、コアデバイスにデプロイしたシークレットの値を出力します。

> **Important**
>
> このコンポーネントの例はシークレットの値を出力するため、テストデータが保存されたシークレットのみで使用してください。このコンポーネントを使用して、重要な情報が保存されたシークレットの値を出力しないでください。

トピック

レシピ

以下のレシピの例はシークレット ARN 設定パラメータを定義し、コンポーネントがコアデバイスのすべてのシークレット値を取得できるようにします。

> **Note**
>
> 実稼働環境では、コンポーネントが使用するシークレットのみ取得するように、承認ポリシーの範囲を小さくすることをお勧めします。コンポーネントをデプロイするとき、* ワイルドカードをシークレット ARN のリストに変更できます。

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PrintSecret",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Prints the value of an AWS Secrets Manager secret.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.SecretManager": {
      "VersionRequirement": "^2.0.0",
      "DependencyType": "HARD"
    }
  },
  "ComponentConfiguration": {
```

```
    "DefaultConfiguration": {
      "SecretArn": "",
      "accessControl": {
        "aws.greengrass.SecretManager": {
          "com.example.PrintSecret:secrets:1": {
            "policyDescription": "Allows access to a secret.",
            "operations": [
              "aws.greengrass#GetSecretValue"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "python3 -m pip install --user awsiotsdk",
        "run": "python3 -u {artifacts:path}/print_secret.py \"{configuration:/
SecretArn}\""
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "py -3 -m pip install --user awsiotsdk",
        "run": "py -3 -u {artifacts:path}/print_secret.py \"{configuration:/
SecretArn}\""
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PrintSecret
ComponentVersion: 1.0.0
ComponentDescription: Prints the value of a Secrets Manager secret.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.SecretManager:
    VersionRequirement: "^2.0.0"
    DependencyType: HARD
ComponentConfiguration:
  DefaultConfiguration:
    SecretArn: ''
    accessControl:
      aws.greengrass.SecretManager:
        com.example.PrintSecret:secrets:1:
          policyDescription: Allows access to a secret.
          operations:
            - aws.greengrass#GetSecretValue
          resources:
            - "*"
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: python3 -m pip install --user awsiotsdk
      run: python3 -u {artifacts:path}/print_secret.py "{configuration:/SecretArn}"
  - Platform:
      os: windows
    Lifecycle:
      install: py -3 -m pip install --user awsiotsdk
      run: py -3 -u {artifacts:path}/print_secret.py "{configuration:/SecretArn}"
```

アーティファクト

次の Python アプリケーションの例は、シークレットマネージャー IPC サービスを使用して、コアデバイスのシークレット値を取得する方法を示しています。

```
import concurrent.futures
import json
```

```
import sys
import traceback

import awsiot.greengrasscoreipc
from awsiot.greengrasscoreipc.model import (
    GetSecretValueRequest,
    GetSecretValueResponse,
    UnauthorizedError
)

TIMEOUT = 10

if len(sys.argv) == 1:
    print('Provide SecretArn in the component configuration.', file=sys.stdout)
    exit(1)

secret_id = sys.argv[1]

try:
    ipc_client = awsiot.greengrasscoreipc.connect()

    request = GetSecretValueRequest()
    request.secret_id = secret_id
    operation = ipc_client.new_get_secret_value()
    operation.activate(request)
    future_response = operation.get_response()

    try:
        response = future_response.result(TIMEOUT)
        secret_json = json.loads(response.secret_value.secret_string)
        print('Successfully got secret: ' + secret_id)
        print('Secret value: ' + str(secret_json))
    except concurrent.futures.TimeoutError:
        print('Timeout occurred while getting secret: ' + secret_id, file=sys.stderr)
    except UnauthorizedError as e:
        print('Unauthorized error while getting secret: ' + secret_id,
 file=sys.stderr)
        raise e
    except Exception as e:
        print('Exception while getting secret: ' + secret_id, file=sys.stderr)
        raise e
except Exception:
    print('Exception occurred when using IPC.', file=sys.stderr)
    traceback.print_exc()
```

```
    exit(1)
```

使用方法

このコンポーネントの例をシークレットマネージャーコンポーネントと一緒に使用すると、コアデバイスのシークレットの値をデプロイおよび出力できます。

テストシークレットを作成、デプロイ、および出力するには

1. テストデータを使用して、Secrets Manager シークレットを作成します。

   Linux or Unix

   ```
   aws secretsmanager create-secret \
     --name MyTestGreengrassSecret \
     --secret-string '{"my-secret-key": "my-secret-value"}'
   ```

   Windows Command Prompt (CMD)

   ```
   aws secretsmanager create-secret ^
     --name MyTestGreengrassSecret ^
     --secret-string '{"my-secret-key": "my-secret-value"}'
   ```

   PowerShell

   ```
   aws secretsmanager create-secret `
     --name MyTestGreengrassSecret `
     --secret-string '{"my-secret-key": "my-secret-value"}'
   ```

   次の手順で使用するシークレットの ARN を保存します。

   詳細については、「AWS Secrets Manager ユーザーガイド」の「シークレットの作成」を参照してください。

2. 以下の設定マージの更新を使用して、シークレットマネージャーコンポーネント (`aws.greengrass.SecretManager`) をデプロイします。先に作成したシークレットの ARN を指定します。

   ```
   {
     "cloudSecrets": [
   ```

```
    {
      "arn": "arn:aws:secretsmanager:us-
west-2:123456789012:secret:MyTestGreengrassSecret-abcdef"
    }
  ]
}
```

詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」または「Greengrass CLI デプロイコマンド」を参照してください。

3. 以下の設定マージの更新を使用して、このセクションのコンポーネントの例を作成してデプロイします。先に作成したシークレットの ARN を指定します。

```
{
  "SecretArn": "arn:aws:secretsmanager:us-
west-2:123456789012:secret:MyTestGreengrassSecret",
  "accessControl": {
    "aws.greengrass.SecretManager": {
      "com.example.PrintSecret:secrets:1": {
        "policyDescription": "Allows access to a secret.",
        "operations": [
          "aws.greengrass#GetSecretValue"
        ],
        "resources": [
          "arn:aws:secretsmanager:us-
west-2:123456789012:secret:MyTestGreengrassSecret-abcdef"
        ]
      }
    }
  }
}
```

詳細については、「AWS IoT Greengrass コンポーネントの作成」を参照してください

4. AWS IoT Greengrass Core ソフトウェアログを表示してデプロイが成功したことを確認し、com.example.PrintSecret コンポーネントログを表示してシークレット値が出力されたことを確認します。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

ローカルシャドウとやり取り

シャドウ IPC サービスを使用して、デバイスのローカルシャドウとやり取りします。やり取りするデバイスには、コアデバイスまたは接続されたクライアントデバイスを選択できます。

これらの IPC オペレーションを使用するには、シャドウマネージャーコンポーネントをカスタムコンポーネントの依存関係として含めます。その後、カスタムコンポーネントの IPC オペレーションを使用して、シャドウマネージャーを介してデバイスのローカルシャドウとやり取りできます。カスタムコンポーネントがローカルシャドウの状態の変更に対応できるようにするには、パブリッシュ/サブスクライブ IPC サービスを使用して、シャドウイベントをサブスクライブすることもできます。パブリッシュ/サブスクライブサービスの使用の詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。

> **Note**
>
> コアデバイスがクライアントデバイスシャドウとやり取りできるようにするには、MQTT ブリッジコンポーネントを設定してデプロイする必要もあります。詳細については、「Enable shadow manager to communicate with client devices」(シャドウマネージャーがクライアントデバイスと通信できるようにする) を参照してください。

トピック

最小 SDK バージョン

以下の表に AWS IoT Device SDK の最小バージョンを示します。ローカルシャドウとやり取りする際は、これを使用する必要があります。

SDK	最小バージョン	
AWS IoT Device SDK for Java v2	v1.4.0	
AWS IoT Device SDK for Python v2	v1.6.0	
AWS IoT Device SDK for C++ v2	v1.17.0	
AWS IoT Device SDK JavaScript v2 用	v1.12.0	

認証

カスタムコンポーネントでシャドウ IPC サービスを使用するには、コンポーネントがシャドウとやり取りできるように承認ポリシーを定義する必要があります。承認ポリシーの定義については、「コンポーネントに IPC オペレーションの実行を許可する」を参照してください。

シャドウ操作の承認ポリシーには以下のプロパティがあります。

IPC サービス識別子: `aws.greengrass.ShadowManager`

操作	説明	リソース
`aws.greengrass#GetThingShadow`	コンポーネントがモノのシャドウを取得できるようにします。	次の文字列のいずれか。 • `$aws/things/`*`thingName`*`/shadow/` は、クラシックデバイスシャドウへのアクセスを許可します。 • `$aws/things/`*`thingName`*`/shadow/name/`*`shadowName`* は、

操作	説明	リソース
		名前付きシャドウへのアクセスを許可します。 • * は、すべてのシャドウへのアクセスを許可します。
aws.greengrass#UpdateThingShadow	コンポーネントがモノのシャドウを更新できるようにします。	次の文字列のいずれか。 • $aws/things/ *thingName* /shadow/ は、クラシックデバイスシャドウへのアクセスを許可します。 • $aws/things/ *thingName* /shadow/name/ *shadowName* は、名前付きシャドウへのアクセスを許可します。 • * は、すべてのシャドウへのアクセスを許可します。

操作	説明	リソース
aws.greengrass#DeleteThingShadow	コンポーネントがモノのシャドウを削除できるようにします。	次の文字列のいずれか。 • $aws/things/ *thingName* /shadow/ は、クラシックデバイスシャドウへのアクセスを許可します。 • $aws/things/ *thingName* /shadow/name/ *shadowName* は、名前付きシャドウへのアクセスを許可します。 • * は、すべてのシャドウへのアクセスを許可します。
aws.greengrass#ListNamedShadowsForThing	コンポーネントがモノの名前付きシャドウのリストを取得できるようにします。	モノにアクセスしてそのシャドウを一覧表示できるようにするモノの名前の文字列。 * を使用してすべてのモノへのアクセスを許可します。

IPC サービス識別子: aws.greengrass.ipc.pubsub

操作	説明	リソース
aws.greengrass#SubscribeToTopic	コンポーネントが、指定したトピックに関するメッセージをサブスクライブできるようにします。	次のトピック文字列のいずれか。 • *shadowTopicPrefix* /get/accepted • *shadowTopicPrefix* /get/rejected

操作	説明	リソース
		• *shadowTopicPrefix* / delete/accepted • *shadowTopicPrefix* / delete/rejected • *shadowTopicPrefix* / update/accepted • *shadowTopicPrefix* / update/delta • *shadowTopicPrefix* / update/rejected トピック接頭辞 *shadowTopicPrefix* の値は、以下に挙げるシャドウのタイプに応じて変化します。 • クラシックシャドウ: $aws/things/ *thingName* /shadow • 名前付きシャドウ: $aws/things/ *thingName* /shadow/name/ *shadowName* すべてのトピックへのアクセスを許可するには、* を使用します。 Greengrass nucleus v2.6.0 以降では、MQTT トピックワイルドカード (# および +) を含むトピックをサブスクライブできます。このトピック文字列は MQTT トピックのワ

操作	説明	リソース
		イルドカードを文字そのものとしてサポートします。例えば、コンポーネントの承認ポリシーで `test/topic/#` へのアクセス権が付与されている場合、コンポーネントは `test/topic/#` をサブスクライブできますが、`test/topic/filter` はサブスクライブできません。

ローカルシャドウ承認ポリシーのレシピ変数

Greengrass nucleus の v2.6.0 以降を使用していて、Greengrass nucleus interpolateComponentConfigurationの設定オプションを に設定した場合は`true`、承認ポリシーで `{iot:thingName}` recipe 変数を使用できます。この機能を使用すると、コアデバイスのグループに対して 1 つの承認ポリシーを設定できます。各コアデバイスは自身のシャドウにのみアクセスできます。例えば、シャドウ IPC オペレーションのために、コンポーネントに次のリソースへのアクセスを許可することができます。

```
$aws/things/{iot:thingName}/shadow/
```

承認ポリシーの例

次の承認ポリシーの例を参照して、コンポーネントの承認ポリシー設定の参考にできます。

Example 例: コアデバイスのグループがローカルシャドウとやり取りすることを許可する

> **Important**
>
> この例では、機能は、Greengrass nucleus コンポーネントの v2.6.0 以降で利用できる機能を使用しています。Greengrass nucleus v2.6.0 では、コンポーネント設定に、ほとんどの recipe 変数 (`{iot:thingName}` など) のサポートが追加されました。この機能を有効にするには、Greengrass nucleus interpolateComponentConfigurationの設定オプションを に設定しますtrue。Greengrass nucleus のすべてのバージョンで機能する例については、「example

authorization policy for a single core device」(シングルコアデバイスの承認ポリシーの例) を参照してください。

次の承認ポリシーの例では、コンポーネント com.example.MyShadowInteractionComponent がクラシックデバイスシャドウ、およびコンポーネントを実行しているコアデバイスの名前付きシャドウ myNamedShadow とやり取りできるようにします。このポリシーは、このコンポーネントがこれらのシャドウのローカルトピックに関するメッセージを受信できるようにもします。

JSON

```
{
  "accessControl": {
    "aws.greengrass.ShadowManager": {
      "com.example.MyShadowInteractionComponent:shadow:1": {
        "policyDescription": "Allows access to shadows",
        "operations": [
          "aws.greengrass#GetThingShadow",
          "aws.greengrass#UpdateThingShadow",
          "aws.greengrass#DeleteThingShadow"
        ],
        "resources": [
          "$aws/things/{iot:thingName}/shadow",
          "$aws/things/{iot:thingName}/shadow/name/myNamedShadow"
        ]
      },
      "com.example.MyShadowInteractionComponent:shadow:2": {
        "policyDescription": "Allows access to things with shadows",
        "operations": [
          "aws.greengrass#ListNamedShadowsForThing"
        ],
        "resources": [
          "{iot:thingName}"
        ]
      }
    },
    "aws.greengrass.ipc.pubsub": {
      "com.example.MyShadowInteractionComponent:pubsub:1": {
        "policyDescription": "Allows access to shadow pubsub topics",
        "operations": [
          "aws.greengrass#SubscribeToTopic"
        ],
```

```
        "resources": [
          "$aws/things/{iot:thingName}/shadow/get/accepted",
          "$aws/things/{iot:thingName}/shadow/name/myNamedShadow/get/accepted"
        ]
      }
    }
  }
}
```

YAML

```
accessControl:
  aws.greengrass.ShadowManager:
    'com.example.MyShadowInteractionComponent:shadow:1':
      policyDescription: 'Allows access to shadows'
      operations:
        - 'aws.greengrass#GetThingShadow'
        - 'aws.greengrass#UpdateThingShadow'
        - 'aws.greengrass#DeleteThingShadow'
      resources:
        - $aws/things/{iot:thingName}/shadow
        - $aws/things/{iot:thingName}/shadow/name/myNamedShadow
    'com.example.MyShadowInteractionComponent:shadow:2':
      policyDescription: 'Allows access to things with shadows'
      operations:
        - 'aws.greengrass#ListNamedShadowsForThing'
      resources:
        - '{iot:thingName}'
  aws.greengrass.ipc.pubsub:
    'com.example.MyShadowInteractionComponent:pubsub:1':
      policyDescription: 'Allows access to shadow pubsub topics'
      operations:
        - 'aws.greengrass#SubscribeToTopic'
      resources:
        - $aws/things/{iot:thingName}/shadow/get/accepted
        - $aws/things/{iot:thingName}/shadow/name/myNamedShadow/get/accepted
```

Example 例: コアデバイスのグループがクライアントデバイスシャドウとやり取りすることを許可する

> **Important**
>
> この機能には、Greengrass nucleus v2.6.0 以降、シャドウマネージャー v2.2.0 以降、および MQTT ブリッジ v2.2.0 以降が必要です。シャドウマネージャーがクライアントデバイスと通信できるように MQTT ブリッジを設定する必要があります。

次の承認ポリシーの例では、コンポーネント `com.example.MyShadowInteractionComponent` が、名前が `MyClientDevice` で始まるクライアントデバイスのすべてのデバイスシャドウとやり取りできるようにします。

> **Note**
>
> コアデバイスがクライアントデバイスシャドウとやり取りできるようにするには、MQTT ブリッジコンポーネントを設定してデプロイする必要もあります。詳細については、「Enable shadow manager to communicate with client devices」(シャドウマネージャーがクライアントデバイスと通信できるようにする) を参照してください。

JSON

```
{
  "accessControl": {
    "aws.greengrass.ShadowManager": {
      "com.example.MyShadowInteractionComponent:shadow:1": {
        "policyDescription": "Allows access to shadows",
        "operations": [
          "aws.greengrass#GetThingShadow",
          "aws.greengrass#UpdateThingShadow",
          "aws.greengrass#DeleteThingShadow"
        ],
        "resources": [
          "$aws/things/MyClientDevice*/shadow",
          "$aws/things/MyClientDevice*/shadow/name/*"
        ]
      },
      "com.example.MyShadowInteractionComponent:shadow:2": {
        "policyDescription": "Allows access to things with shadows",
```

```
        "operations": [
          "aws.greengrass#ListNamedShadowsForThing"
        ],
        "resources": [
          "MyClientDevice*"
        ]
      }
    }
  }
}
```

YAML

```
accessControl:
  aws.greengrass.ShadowManager:
    'com.example.MyShadowInteractionComponent:shadow:1':
      policyDescription: 'Allows access to shadows'
      operations:
        - 'aws.greengrass#GetThingShadow'
        - 'aws.greengrass#UpdateThingShadow'
        - 'aws.greengrass#DeleteThingShadow'
      resources:
        - $aws/things/MyClientDevice*/shadow
        - $aws/things/MyClientDevice*/shadow/name/*
    'com.example.MyShadowInteractionComponent:shadow:2':
      policyDescription: 'Allows access to things with shadows'
      operations:
        - 'aws.greengrass#ListNamedShadowsForThing'
      resources:
        - MyClientDevice*
```

Example 例: シングルコアデバイスがローカルシャドウとやり取りすることを許可する

以下の承認ポリシーの例は、コンポーネント `com.example.MyShadowInteractionComponent` がクラシックデバイスシャドウ、およびデバイス `MyThingName` の名前付きシャドウ `myNamedShadow` とやり取りできるようにします。このポリシーは、このコンポーネントがこれらのシャドウのローカルトピックに関するメッセージを受信できるようにもします。

JSON

```
{
```

```
  "accessControl": {
    "aws.greengrass.ShadowManager": {
      "com.example.MyShadowInteractionComponent:shadow:1": {
        "policyDescription": "Allows access to shadows",
        "operations": [
          "aws.greengrass#GetThingShadow",
          "aws.greengrass#UpdateThingShadow",
          "aws.greengrass#DeleteThingShadow"
        ],
        "resources": [
          "$aws/things/MyThingName/shadow",
          "$aws/things/MyThingName/shadow/name/myNamedShadow"
        ]
      },
      "com.example.MyShadowInteractionComponent:shadow:2": {
        "policyDescription": "Allows access to things with shadows",
        "operations": [
          "aws.greengrass#ListNamedShadowsForThing"
        ],
        "resources": [
          "MyThingName"
        ]
      }
    },
    "aws.greengrass.ipc.pubsub": {
      "com.example.MyShadowInteractionComponent:pubsub:1": {
        "policyDescription": "Allows access to shadow pubsub topics",
        "operations": [
          "aws.greengrass#SubscribeToTopic"
        ],
        "resources": [
          "$aws/things/MyThingName/shadow/get/accepted",
          "$aws/things/MyThingName/shadow/name/myNamedShadow/get/accepted"
        ]
      }
    }
  }
}
```

YAML

```
accessControl:
  aws.greengrass.ShadowManager:
```

```
    'com.example.MyShadowInteractionComponent:shadow:1':
      policyDescription: 'Allows access to shadows'
      operations:
        - 'aws.greengrass#GetThingShadow'
        - 'aws.greengrass#UpdateThingShadow'
        - 'aws.greengrass#DeleteThingShadow'
      resources:
        - $aws/things/MyThingName/shadow
        - $aws/things/MyThingName/shadow/name/myNamedShadow
    'com.example.MyShadowInteractionComponent:shadow:2':
      policyDescription: 'Allows access to things with shadows'
      operations:
        - 'aws.greengrass#ListNamedShadowsForThing'
      resources:
        - MyThingName
  aws.greengrass.ipc.pubsub:
    'com.example.MyShadowInteractionComponent:pubsub:1':
      policyDescription: 'Allows access to shadow pubsub topics'
      operations:
        - 'aws.greengrass#SubscribeToTopic'
      resources:
        - $aws/things/MyThingName/shadow/get/accepted
        - $aws/things/MyThingName/shadow/name/myNamedShadow/get/accepted
```

Example 例: コアデバイスのグループがローカルシャドウの状態変化に反応することを許可する

> **Important**
>
> この例では、機能は、Greengrass nucleus コンポーネントの v2.6.0 以降で利用できる機能を使用しています。Greengrass nucleus v2.6.0 では、コンポーネント設定に、ほとんどの recipe 変数 (`{iot:thingName}` など) のサポートが追加されました。この機能を有効にするには、Greengrass nucleus interpolateComponentConfigurationの設定オプションを に設定しますtrue。Greengrass nucleus のすべてのバージョンで機能する例については、「example authorization policy for a single core device」(シングルコアデバイスの承認ポリシーの例) を参照してください。

次のアクセスコントロールポリシーの例では、クラシックデバイスシャドウおよび名前付きシャドウ myNamedShadow の `/update/delta` トピックに関するメッセージを、コンポーネントを実行して

いる各コアデバイスで、カスタム com.example.MyShadowReactiveComponent が受信できるようにします。

JSON

```
{
  "accessControl": {
    "aws.greengrass.ipc.pubsub": {
      "com.example.MyShadowReactiveComponent:pubsub:1": {
        "policyDescription": "Allows access to shadow pubsub topics",
        "operations": [
          "aws.greengrass#SubscribeToTopic"
        ],
        "resources": [
          "$aws/things/{iot:thingName}/shadow/update/delta",
          "$aws/things/{iot:thingName}/shadow/name/myNamedShadow/update/delta"
        ]
      }
    }
  }
}
```

YAML

```
accessControl:
  aws.greengrass.ipc.pubsub:
    "com.example.MyShadowReactiveComponent:pubsub:1":
      policyDescription: Allows access to shadow pubsub topics
      operations:
        - 'aws.greengrass#SubscribeToTopic'
      resources:
        - $aws/things/{iot:thingName}/shadow/update/delta
        - $aws/things/{iot:thingName}/shadow/name/myNamedShadow/update/delta
```

Example 例: シングルコアデバイスに、ローカルシャドウの状態の変化に反応することを許可する

次のアクセスコントロールポリシーの例では、デバイス MyThingName について、クラシックデバイスシャドウおよび名前付きシャドウ myNamedShadow の /update/delta トピックに関するメッセージを、カスタム com.example.MyShadowReactiveComponent が受信できるようにします。

JSON

```
{
  "accessControl": {
    "aws.greengrass.ipc.pubsub": {
      "com.example.MyShadowReactiveComponent:pubsub:1": {
        "policyDescription": "Allows access to shadow pubsub topics",
        "operations": [
          "aws.greengrass#SubscribeToTopic"
        ],
        "resources": [
          "$aws/things/MyThingName/shadow/update/delta",
          "$aws/things/MyThingName/shadow/name/myNamedShadow/update/delta"
        ]
      }
    }
  }
}
```

YAML

```
accessControl:
  aws.greengrass.ipc.pubsub:
    "com.example.MyShadowReactiveComponent:pubsub:1":
      policyDescription: Allows access to shadow pubsub topics
      operations:
        - 'aws.greengrass#SubscribeToTopic'
      resources:
        - $aws/things/MyThingName/shadow/update/delta
        - $aws/things/MyThingName/shadow/name/myNamedShadow/update/delta
```

GetThingShadow

指定したモノのシャドウを取得します。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`thingName` (Python: `thing_name`)

モノの名前。

タイプ: `string`

`shadowName` (Python: `shadow_name`)

シャドウの名前。モノのクラシックシャドウを指定するには、このパラメータを空の文字列 ("") に設定します。

> **Warning**
>
> AWS IoT Greengrass サービスは、`AWSManagedGreengrassV2Deployment` 名前付きシャドウを使用して、個々のコアデバイスを対象とするデプロイを管理します。この名前付きシャドウは、AWS IoT Greengrass サービスで使用するために予約されています。この名前付きシャドウを更新または削除しないでください。

タイプ: `string`

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`payload`

BLOB としてのレスポンス状態ドキュメント。

タイプ: 次の情報が含まれる `object`。

`state`

状態情報。

このオブジェクトには、次の情報が含まれます。

`desired`

デバイスで更新がリクエストされた state のプロパティと値。

タイプ: キーバリューペアの `map`

`reported`

デバイスによってレポートされた state のプロパティと値。

タイプ: キーバリューペアの `map`

`delta`

望ましい state とレポートされた state のプロパティと値の違い。このプロパティは、`desired` と `reported` の state が異なる場合のみ存在します。

タイプ: キーバリューペアの `map`

`metadata`

いつ状態が更新されたか判別するための、`desired` および `reported` セクションの属性ごとのタイムスタンプ。

タイプ: `string`

`timestamp`

レスポンスが生成された日付と時刻 (エポック時間)。

タイプ: `integer`

`clientToken` (Python: `clientToken`)

リクエストとレスポンスを対応付けるために使用されるトークン

タイプ: `string`

`version`

ローカルシャドウドキュメントのバージョン。

タイプ: `integer`

エラー

このオペレーションは以下のエラーを返す場合があります。

`InvalidArgumentsError`

ローカルシャドウサービスは、リクエストパラメータを検証できません。これは、リクエストに不正な形式の JSON またはサポートされていない文字が含まれている場合に発生する可能性があります。

`ResourceNotFoundError`

要求されたローカルシャドウドキュメントが見つかりません。

`ServiceError`

内部サービスエラーが発生したか、IPC サービスへのリクエスト数が、シャドウマネージャーコンポーネントの `maxLocalRequestsPerSecondPerThing` および `maxTotalLocalRequestsRate` の設定パラメータで指定された制限を超えました。

`UnauthorizedError`

コンポーネントの承認ポリシーには、このオペレーションに必要な権限が含まれていません。

例

以下の例では、カスタムコンポーネントコードでこのオペレーションを呼び出す方法を示します。

Java (IPC client V1)

Example 例: モノのシャドウを取得する

> **Note**
>
> この例は `IPCUtils` クラスを使用して、AWS IoT Greengrass Core IPC サービスへの接続を作成します。詳細については、「AWS IoT Greengrass Core IPC サービスに接続する」を参照してください。

```
package com.aws.greengrass.docs.samples.ipc;

import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.GetThingShadowResponseHandler;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.model.GetThingShadowRequest;
import software.amazon.awssdk.aws.greengrass.model.GetThingShadowResponse;
import software.amazon.awssdk.aws.greengrass.model.ResourceNotFoundError;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;

import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
```

```
public class GetThingShadow {

    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        // Use the current core device's name if thing name isn't set.
        String thingName = args[0].isEmpty() ? System.getenv("AWS_IOT_THING_NAME") :
 args[0];
        String shadowName = args[1];
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            GetThingShadowResponseHandler responseHandler =
                    GetThingShadow.getThingShadow(ipcClient, thingName,
 shadowName);
            CompletableFuture<GetThingShadowResponse> futureResponse =
                    responseHandler.getResponse();
            try {
                GetThingShadowResponse response =
 futureResponse.get(TIMEOUT_SECONDS,
                        TimeUnit.SECONDS);
                String shadowPayload = new String(response.getPayload(),
 StandardCharsets.UTF_8);
                System.out.printf("Successfully got shadow %s/%s: %s%n", thingName,
 shadowName,
                        shadowPayload);
            } catch (TimeoutException e) {
                System.err.printf("Timeout occurred while getting shadow: %s/%s%n",
 thingName,
                        shadowName);
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.printf("Unauthorized error while getting shadow: %s/
%s%n",
                            thingName, shadowName);
                } else if (e.getCause() instanceof ResourceNotFoundError) {
                    System.err.printf("Unable to find shadow to get: %s/%s%n",
 thingName,
                            shadowName);
                } else {
                    throw e;
                }
```

```
            }
        } catch (InterruptedException e) {
            System.out.println("IPC interrupted.");
        } catch (ExecutionException e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static GetThingShadowResponseHandler
 getThingShadow(GreengrassCoreIPCClient greengrassCoreIPCClient, String thingName,
 String shadowName) {
        GetThingShadowRequest getThingShadowRequest = new GetThingShadowRequest();
        getThingShadowRequest.setThingName(thingName);
        getThingShadowRequest.setShadowName(shadowName);
        return greengrassCoreIPCClient.getThingShadow(getThingShadowRequest,
 Optional.empty());
    }
}
```

Python (IPC client V1)

Example 例: モノのシャドウを取得する

```
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import GetThingShadowRequest

TIMEOUT = 10

def sample_get_thing_shadow_request(thingName, shadowName):
    try:
        # set up IPC client to connect to the IPC server
        ipc_client = awsiot.greengrasscoreipc.connect()

        # create the GetThingShadow request
        get_thing_shadow_request = GetThingShadowRequest()
        get_thing_shadow_request.thing_name = thingName
        get_thing_shadow_request.shadow_name = shadowName

        # retrieve the GetThingShadow response after sending the request to the IPC
 server
```

```
        op = ipc_client.new_get_thing_shadow()
        op.activate(get_thing_shadow_request)
        fut = op.get_response()

        result = fut.result(TIMEOUT)
        return result.payload

    except InvalidArgumentsError as e:
        # add error handling
        ...
    # except ResourceNotFoundError | UnauthorizedError | ServiceError
```

JavaScript

Example 例: モノのシャドウを取得する

```
import {
    GetThingShadowRequest
} from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc/model';
import * as greengrasscoreipc from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc';

class GetThingShadow {
    private ipcClient: greengrasscoreipc.Client;
    private thingName: string;
    private shadowName: string;

    constructor() {
        // Define args parameters here
        this.thingName = "<define_your_own_thingName>";
        this.shadowName = "<define_your_own_shadowName>";
        this.bootstrap();
    }

    async bootstrap() {
        try {
            this.ipcClient = await getIpcClient();
        } catch (err) {
            // parse the error depending on your use cases
            throw err
        }

        try {
            await this.handleGetThingShadowOperation(this.thingName,
                this.shadowName);
```

```
        } catch (err) {
            // parse the error depending on your use cases
            throw err
        }
    }

    async handleGetThingShadowOperation(
        thingName: string,
        shadowName: string
    ) {
        const request: GetThingShadowRequest = {
            thingName: thingName,
            shadowName: shadowName
        };
        const response = await this.ipcClient.getThingShadow(request);
    }
}

export async function getIpcClient() {
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use caseså
        throw err
    }
}

const startScript = new GetThingShadow();
```

UpdateThingShadow

指定したモノのシャドウを更新します。Shadow が存在しない場合は作成されます。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`thingName` (Python: `thing_name`)

モノの名前。

タイプ: `string`

`shadowName` (Python: `shadow_name`)

シャドウの名前。モノのクラシックシャドウを指定するには、このパラメータを空の文字列 ("") に設定します。

> ⚠ Warning
>
> AWS IoT Greengrass サービスは、`AWSManagedGreengrassV2Deployment` 名前付きシャドウを使用して、個々のコアデバイスを対象とするデプロイを管理します。この名前付きシャドウは、AWS IoT Greengrass サービスで使用するために予約されています。この名前付きシャドウを更新または削除しないでください。

タイプ: `string`

`payload`

BLOB としてのリクエスト状態ドキュメント。

タイプ: 次の情報が含まれる `object`。

`state`

更新する状態情報。この IPC オペレーションは、指定したフィールドのみに影響します。

このオブジェクトには、次の情報が含まれます。通常、同じリクエストで `desired` プロパティまたは `reported` プロパティのいずれかを使用しますが、両方を使用することはありません。

`desired`

デバイスで更新がリクエストされた state のプロパティと値。

タイプ: キーバリューペアの `map`

`reported`

デバイスによってレポートされた state のプロパティと値。

タイプ: キーバリューペアの `map`

`clientToken` (Python: `client_token`)

(オプション) クライアントトークンによってリクエストとレスポンスを対応付けるために使用されるトークン。

タイプ: `string`

`version`

(オプション) 更新するローカルシャドウドキュメントのバージョン。シャドウサービスは、指定したバージョンが最新バージョンと一致した場合のみ更新を処理します。

タイプ: `integer`

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`payload`

BLOB としてのレスポンス状態ドキュメント。

タイプ: 次の情報が含まれる `object`。

`state`

状態情報。

このオブジェクトには、次の情報が含まれます。

`desired`

デバイスで更新がリクエストされた state のプロパティと値。

タイプ: キーバリューペアの `map`

`reported`

デバイスによってレポートされた state のプロパティと値。

タイプ: キーバリューペアの `map`

`delta`

デバイスによってレポートされた state のプロパティと値。

タイプ: キーバリューペアの `map`

`metadata`

いつ状態が更新されたか判別するための、`desired` および `reported` セクションの属性ごとのタイムスタンプ。

タイプ: `string`

`timestamp`

レスポンスが生成された日付と時刻 (エポック時間)。

タイプ: `integer`

`clientToken` (Python: `client_token`)

リクエストとレスポンスを対応付けるために使用されるトークン。

タイプ: `string`

`version`

更新完了後のローカルシャドウドキュメントのバージョン。

タイプ: `integer`

エラー

このオペレーションは以下のエラーを返す場合があります。

`ConflictError`

ローカルシャドウサービスで、更新オペレーション中にバージョンの競合が発生しました。これは、リクエストペイロードのバージョンが利用可能な最新のローカルシャドードキュメントのバージョンと一致しない場合に発生します。

`InvalidArgumentsError`

ローカルシャドウサービスは、リクエストパラメータを検証できません。これは、リクエストに不正な形式の JSON またはサポートされていない文字が含まれている場合に発生する可能性があります。

有効な `payload` は以下のプロパティを有します。

- `state` ノードが存在すること、そして `desired` または `reported` の状態情報を含むオブジェクトであること。

- desired および reported ノードはオブジェクトまたはヌルのいずれかであること。これらのオブジェクトの少なくとも 1 つに有効な状態情報が含まれている必要があります。
- desired および reported オブジェクトの深さが 8 ノードを超えることはできません。
- clientToken の値の長さが 64 文字を超えることはできません。
- version の値は 1 以上である必要があります。

ServiceError

内部サービスエラーが発生したか、IPC サービスへのリクエスト数が、シャドウマネージャーコンポーネントの maxLocalRequestsPerSecondPerThing および maxTotalLocalRequestsRate の設定パラメータで指定された制限を超えました。

UnauthorizedError

コンポーネントの承認ポリシーには、このオペレーションに必要な権限が含まれていません。

例

以下の例では、カスタムコンポーネントコードでこのオペレーションを呼び出す方法を示します。

Java (IPC client V1)

Example 例: モノのシャドウを更新する

> **Note**
>
> この例は IPCUtils クラスを使用して、AWS IoT Greengrass Core IPC サービスへの接続を作成します。詳細については、「AWS IoT Greengrass Core IPC サービスに接続する」を参照してください。

```
package com.aws.greengrass.docs.samples.ipc;

import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.UpdateThingShadowResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.aws.greengrass.model.UpdateThingShadowRequest;
import software.amazon.awssdk.aws.greengrass.model.UpdateThingShadowResponse;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;
```

```
import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class UpdateThingShadow {

    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        // Use the current core device's name if thing name isn't set.
        String thingName = args[0].isEmpty() ? System.getenv("AWS_IOT_THING_NAME") :
 args[0];
        String shadowName = args[1];
        byte[] shadowPayload = args[2].getBytes(StandardCharsets.UTF_8);
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            UpdateThingShadowResponseHandler responseHandler =
                    UpdateThingShadow.updateThingShadow(ipcClient, thingName,
 shadowName,
                            shadowPayload);
            CompletableFuture<UpdateThingShadowResponse> futureResponse =
                    responseHandler.getResponse();
            try {
                futureResponse.get(TIMEOUT_SECONDS, TimeUnit.SECONDS);
                System.out.printf("Successfully updated shadow: %s/%s%n", thingName,
 shadowName);
            } catch (TimeoutException e) {
                System.err.printf("Timeout occurred while updating shadow: %s/%s%n",
 thingName,
                        shadowName);
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.printf("Unauthorized error while updating shadow: %s/
%s%n",
                            thingName, shadowName);
                } else {
                    throw e;
                }
            }
```

```
        } catch (InterruptedException e) {
            System.out.println("IPC interrupted.");
        } catch (ExecutionException e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static UpdateThingShadowResponseHandler
 updateThingShadow(GreengrassCoreIPCClient greengrassCoreIPCClient, String
 thingName, String shadowName, byte[] shadowPayload) {
        UpdateThingShadowRequest updateThingShadowRequest = new
 UpdateThingShadowRequest();
        updateThingShadowRequest.setThingName(thingName);
        updateThingShadowRequest.setShadowName(shadowName);
        updateThingShadowRequest.setPayload(shadowPayload);
        return greengrassCoreIPCClient.updateThingShadow(updateThingShadowRequest,
                Optional.empty());
    }
}
```

Python (IPC client V1)

Example 例: モノのシャドウを更新する

```
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import UpdateThingShadowRequest

TIMEOUT = 10

def sample_update_thing_shadow_request(thingName, shadowName, payload):
    try:
        # set up IPC client to connect to the IPC server
        ipc_client = awsiot.greengrasscoreipc.connect()

        # create the UpdateThingShadow request
        update_thing_shadow_request = UpdateThingShadowRequest()
        update_thing_shadow_request.thing_name = thingName
        update_thing_shadow_request.shadow_name = shadowName
        update_thing_shadow_request.payload = payload
```

```
        # retrieve the UpdateThingShadow response after sending the request to the
 IPC server
        op = ipc_client.new_update_thing_shadow()
        op.activate(update_thing_shadow_request)
        fut = op.get_response()

        result = fut.result(TIMEOUT)
        return result.payload

    except InvalidArgumentsError as e:
        # add error handling
    ...
    # except ConflictError | UnauthorizedError | ServiceError
```

JavaScript

Example 例: モノのシャドウを更新する

```
import {
    UpdateThingShadowRequest
} from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc/model';
import * as greengrasscoreipc from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc';

class UpdateThingShadow {
    private ipcClient: greengrasscoreipc.Client;
    private thingName: string;
    private shadowName: string;
    private shadowDocumentStr: string;

    constructor() {
        // Define args parameters here

        this.thingName = "<define_your_own_thingName>";
        this.shadowName = "<define_your_own_shadowName>";
        this.shadowDocumentStr = "<define_your_own_payload>";

        this.bootstrap();
    }

    async bootstrap() {
        try {
            this.ipcClient = await getIpcClient();
        } catch (err) {
            // parse the error depending on your use cases
```

```
            throw err
        }

        try {
            await this.handleUpdateThingShadowOperation(
                this.thingName,
                this.shadowName,
                this.shadowDocumentStr);
        } catch (err) {
            // parse the error depending on your use cases
            throw err
        }
    }

    async handleUpdateThingShadowOperation(
        thingName: string,
        shadowName: string,
        payloadStr: string
    ) {
        const request: UpdateThingShadowRequest = {
            thingName: thingName,
            shadowName: shadowName,
            payload: payloadStr
        }
        // make the UpdateThingShadow request
        const response = await this.ipcClient.updateThingShadow(request);
    }
}

export async function getIpcClient() {
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}
```

```
const startScript = new UpdateThingShadow();
```

DeleteThingShadow

指定したモノの Shadow を削除します。

シャドウマネージャー v2.0.4 以降、シャドウを削除するとバージョン番号がインクリメントします。たとえば、バージョン 1 のシャドウ MyThingShadow を削除すると、削除されたシャドウのバージョンは 2 になります。その後 MyThingShadow の名前でシャドウを作成し直すと、そのシャドウのバージョンは 3 になります。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

thingName (Python: thing_name)

モノの名前。

タイプ: string

shadowName (Python: shadow_name)

シャドウの名前。モノのクラシックシャドウを指定するには、このパラメータを空の文字列 ("") に設定します。

> **Warning**
>
> AWS IoT Greengrass サービスは、AWSManagedGreengrassV2Deployment 名前付きシャドウを使用して、個々のコアデバイスを対象とするデプロイを管理します。この名前付きシャドウは、AWS IoT Greengrass サービスで使用するために予約されています。この名前付きシャドウを更新または削除しないでください。

タイプ: string

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`payload`

空のレスポンス状態ドキュメント。

エラー

このオペレーションは以下のエラーを返す場合があります。

`InvalidArgumentsError`

ローカルシャドウサービスは、リクエストパラメータを検証できません。これは、リクエストに不正な形式の JSON またはサポートされていない文字が含まれている場合に発生する可能性があります。

`ResourceNotFoundError`

要求されたローカルシャドウドキュメントが見つかりません。

`ServiceError`

内部サービスエラーが発生したか、IPC サービスへのリクエスト数が、シャドウマネージャーコンポーネントの `maxLocalRequestsPerSecondPerThing` および `maxTotalLocalRequestsRate` の設定パラメータで指定された制限を超えました。

`UnauthorizedError`

コンポーネントの承認ポリシーには、このオペレーションに必要な権限が含まれていません。

例

以下の例では、カスタムコンポーネントコードでこのオペレーションを呼び出す方法を示します。

Java (IPC client V1)

Example 例: モノのシャドウを削除する

> **Note**
>
> この例は `IPCUtils` クラスを使用して、AWS IoT Greengrass Core IPC サービスへの接続を作成します。詳細については、「AWS IoT Greengrass Core IPC サービスに接続する」を参照してください。

```
package com.aws.greengrass.docs.samples.ipc;

import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.DeleteThingShadowResponseHandler;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.model.DeleteThingShadowRequest;
import software.amazon.awssdk.aws.greengrass.model.DeleteThingShadowResponse;
import software.amazon.awssdk.aws.greengrass.model.ResourceNotFoundError;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;

import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class DeleteThingShadow {

    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        // Use the current core device's name if thing name isn't set.
        String thingName = args[0].isEmpty() ? System.getenv("AWS_IOT_THING_NAME") :
 args[0];
        String shadowName = args[1];
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            DeleteThingShadowResponseHandler responseHandler =
                    DeleteThingShadow.deleteThingShadow(ipcClient, thingName,
 shadowName);
            CompletableFuture<DeleteThingShadowResponse> futureResponse =
                    responseHandler.getResponse();
            try {
                futureResponse.get(TIMEOUT_SECONDS, TimeUnit.SECONDS);
                System.out.printf("Successfully deleted shadow: %s/%s%n", thingName,
 shadowName);
            } catch (TimeoutException e) {
                System.err.printf("Timeout occurred while deleting shadow: %s/%s%n",
 thingName,
                        shadowName);
```

```
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.printf("Unauthorized error while deleting shadow: %s/
%s%n",
                            thingName, shadowName);
                } else if (e.getCause() instanceof ResourceNotFoundError) {
                    System.err.printf("Unable to find shadow to delete: %s/%s%n",
 thingName,
                            shadowName);
                } else {
                    throw e;
                }
            }
        } catch (InterruptedException e) {
            System.out.println("IPC interrupted.");
        } catch (ExecutionException e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static DeleteThingShadowResponseHandler
 deleteThingShadow(GreengrassCoreIPCClient greengrassCoreIPCClient, String
 thingName, String shadowName) {
        DeleteThingShadowRequest deleteThingShadowRequest = new
 DeleteThingShadowRequest();
        deleteThingShadowRequest.setThingName(thingName);
        deleteThingShadowRequest.setShadowName(shadowName);
        return greengrassCoreIPCClient.deleteThingShadow(deleteThingShadowRequest,
                Optional.empty());
    }
}
```

Python (IPC client V1)

Example 例: モノのシャドウを削除する

```
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import DeleteThingShadowRequest

TIMEOUT = 10
```

```
def sample_delete_thing_shadow_request(thingName, shadowName):
    try:
        # set up IPC client to connect to the IPC server
        ipc_client = awsiot.greengrasscoreipc.connect()

        # create the DeleteThingShadow request
        delete_thing_shadow_request = DeleteThingShadowRequest()
        delete_thing_shadow_request.thing_name = thingName
        delete_thing_shadow_request.shadow_name = shadowName

        # retrieve the DeleteThingShadow response after sending the request to the
 IPC server
        op = ipc_client.new_delete_thing_shadow()
        op.activate(delete_thing_shadow_request)
        fut = op.get_response()

        result = fut.result(TIMEOUT)
        return result.payload

    except InvalidArgumentsError as e:
        # add error handling
    ...
    # except ResourceNotFoundError | UnauthorizedError | ServiceError
```

JavaScript

Example 例: モノのシャドウを削除する

```
import {
    DeleteThingShadowRequest
} from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc/model';
import * as greengrasscoreipc from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc';

class DeleteThingShadow {
    private ipcClient: greengrasscoreipc.Client;
    private thingName: string;
    private shadowName: string;

    constructor() {
        // Define args parameters here
        this.thingName = "<define_your_own_thingName>";
        this.shadowName = "<define_your_own_shadowName>";
        this.bootstrap();
    }
```

```
    async bootstrap() {
        try {
            this.ipcClient = await getIpcClient();
        } catch (err) {
            // parse the error depending on your use cases
            throw err
        }

        try {
            await this.handleDeleteThingShadowOperation(this.thingName,
 this.shadowName)
        } catch (err) {
            // parse the error depending on your use cases
            throw err
        }
    }

    async handleDeleteThingShadowOperation(thingName: string, shadowName: string) {
        const request: DeleteThingShadowRequest = {
            thingName: thingName,
            shadowName: shadowName
        }
        // make the DeleteThingShadow request
        const response = await this.ipcClient.deleteThingShadow(request);
    }
}

export async function getIpcClient() {
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}
```

```
const startScript = new DeleteThingShadow();
```

ListNamedShadowsForThing

指定されたモノの名前付きシャドウを一覧表示します。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`thingName` (Python: `thing_name`)

モノの名前。

タイプ: `string`

`pageSize` (Python: `page_size`)

(オプション) 各呼び出しで返すシャドウ名の数。

タイプ: `integer`

デフォルト: 25

最大: 100

`nextToken` (Python: `next_token`)

(オプション) 次の結果セットを取得するためのトークン。この値は、ページングされた結果で返され、次のページを返す呼び出しで使用されます。

タイプ: `string`

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`results`

シャドウ名のリスト。

タイプ: `array`

`timestamp`

(オプション) レスポンスが生成された日付と時刻。

タイプ: `integer`

`nextToken` (Python: `next_token`)

(オプション) シーケンス内の次のページを取得するためにページングされたリクエストで使用するトークン値。返すシャドウ名がなくなると、このプロパティは存在しません。

タイプ: `string`

> **Note**
>
> リクエストされたページサイズがレスポンスのシャドウ名の数と完全に一致する場合、このトークンは存在しますが、使用すると空のリストが返されます。

エラー

このオペレーションは以下のエラーを返す場合があります。

`InvalidArgumentsError`

ローカルシャドウサービスは、リクエストパラメータを検証できません。これは、リクエストに不正な形式の JSON またはサポートされていない文字が含まれている場合に発生する可能性があります。

`ResourceNotFoundError`

要求されたローカルシャドウドキュメントが見つかりません。

`ServiceError`

内部サービスエラーが発生したか、IPC サービスへのリクエスト数が、シャドウマネージャーコンポーネントの `maxLocalRequestsPerSecondPerThing` および `maxTotalLocalRequestsRate` の設定パラメータで指定された制限を超えました。

`UnauthorizedError`

コンポーネントの承認ポリシーには、このオペレーションに必要な権限が含まれていません。

例

以下の例では、カスタムコンポーネントコードでこのオペレーションを呼び出す方法を示します。

Java (IPC client V1)

Example 例:モノの名前付きシャドウを一覧表示

> **Note**
>
> この例は `IPCUtils` クラスを使用して、AWS IoT Greengrass Core IPC サービスへの接続を作成します。詳細については、「AWS IoT Greengrass Core IPC サービスに接続する」を参照してください。

```
package com.aws.greengrass.docs.samples.ipc;

import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import
 software.amazon.awssdk.aws.greengrass.ListNamedShadowsForThingResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.ListNamedShadowsForThingRequest;
import
 software.amazon.awssdk.aws.greengrass.model.ListNamedShadowsForThingResponse;
import software.amazon.awssdk.aws.greengrass.model.ResourceNotFoundError;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class ListNamedShadowsForThing {

    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        // Use the current core device's name if thing name isn't set.
```

```
        String thingName = args[0].isEmpty() ? System.getenv("AWS_IOT_THING_NAME") :
 args[0];
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            List<String> namedShadows = new ArrayList<>();
            String nextToken = null;
            try {
                // Send additional requests until there's no pagination token in the
 response.
                do {
                    ListNamedShadowsForThingResponseHandler responseHandler =

 ListNamedShadowsForThing.listNamedShadowsForThing(ipcClient, thingName,
                                    nextToken, 25);
                    CompletableFuture<ListNamedShadowsForThingResponse>
 futureResponse =
                            responseHandler.getResponse();
                    ListNamedShadowsForThingResponse response =
                            futureResponse.get(TIMEOUT_SECONDS, TimeUnit.SECONDS);
                    List<String> responseNamedShadows = response.getResults();
                    namedShadows.addAll(responseNamedShadows);
                    nextToken = response.getNextToken();
                } while (nextToken != null);
                System.out.printf("Successfully got named shadows for thing %s: %s
%n", thingName,
                        String.join(",", namedShadows));
            } catch (TimeoutException e) {
                System.err.println("Timeout occurred while listing named shadows for
 thing: " + thingName);
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.println("Unauthorized error while listing named
 shadows for " +
                            "thing: " + thingName);
                } else if (e.getCause() instanceof ResourceNotFoundError) {
                    System.err.println("Unable to find thing to list named shadows:
 " + thingName);
                } else {
                    throw e;
                }
            }
        } catch (InterruptedException e) {
```

```
            System.out.println("IPC interrupted.");
        } catch (ExecutionException e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static ListNamedShadowsForThingResponseHandler
 listNamedShadowsForThing(GreengrassCoreIPCClient greengrassCoreIPCClient, String
 thingName, String nextToken, int pageSize) {
        ListNamedShadowsForThingRequest listNamedShadowsForThingRequest =
                new ListNamedShadowsForThingRequest();
        listNamedShadowsForThingRequest.setThingName(thingName);
        listNamedShadowsForThingRequest.setNextToken(nextToken);
        listNamedShadowsForThingRequest.setPageSize(pageSize);
        return
 greengrassCoreIPCClient.listNamedShadowsForThing(listNamedShadowsForThingRequest,
                Optional.empty());
    }
}
```

Python (IPC client V1)

Example 例:モノの名前付きシャドウを一覧表示

```
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import ListNamedShadowsForThingRequest

TIMEOUT = 10

def sample_list_named_shadows_for_thing_request(thingName, nextToken, pageSize):
    try:
        # set up IPC client to connect to the IPC server
        ipc_client = awsiot.greengrasscoreipc.connect()

        # create the ListNamedShadowsForThingRequest request
        list_named_shadows_for_thing_request = ListNamedShadowsForThingRequest()
        list_named_shadows_for_thing_request.thing_name = thingName
        list_named_shadows_for_thing_request.next_token = nextToken
        list_named_shadows_for_thing_request.page_size = pageSize
```

```
        # retrieve the ListNamedShadowsForThingRequest response after sending the
 request to the IPC server
        op = ipc_client.new_list_named_shadows_for_thing()
        op.activate(list_named_shadows_for_thing_request)
        fut = op.get_response()

        list_result = fut.result(TIMEOUT)

        # additional returned fields
        timestamp = list_result.timestamp
        next_token = result.next_token
        named_shadow_list = list_result.results

        return named_shadow_list, next_token, timestamp

    except InvalidArgumentsError as e:
        # add error handling
    ...
    # except ResourceNotFoundError | UnauthorizedError | ServiceError
```

JavaScript

Example 例:モノの名前付きシャドウを一覧表示

```
import {
    ListNamedShadowsForThingRequest
} from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc/model';
import * as greengrasscoreipc from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc';

class listNamedShadowsForThing {
    private ipcClient: greengrasscoreipc.Client;
    private thingName: string;
    private pageSizeStr: string;
    private nextToken: string;

    constructor() {
        // Define args parameters here
        this.thingName = "<define_your_own_thingName>";
        this.pageSizeStr = "<define_your_own_pageSize>";
        this.nextToken = "<define_your_own_token>";
        this.bootstrap();
    }

    async bootstrap() {
```

```
        try {
            this.ipcClient = await getIpcClient();
        } catch (err) {
            // parse the error depending on your use cases
            throw err
        }

        try {
            await this.handleListNamedShadowsForThingOperation(this.thingName,
                this.nextToken, this.pageSizeStr);
        } catch (err) {
            // parse the error depending on your use cases
            throw err
        }
    }

    async handleListNamedShadowsForThingOperation(
        thingName: string,
        nextToken: string,
        pageSizeStr: string
    ) {
        let request: ListNamedShadowsForThingRequest = {
            thingName: thingName,
            nextToken: nextToken,
        };
        if (pageSizeStr) {
            request.pageSize = parseInt(pageSizeStr);
        }
        // make the ListNamedShadowsForThing request
        const response = await this.ipcClient.listNamedShadowsForThing(request);
        const shadowNames = response.results;
    }
}

export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
```

```
        // parse the error depending on your use cases
        throw err
    }
}

const startScript = new listNamedShadowsForThing();
```

ローカルデプロイおよびコンポーネントの管理

> **Note**
>
> この機能は、Greengrass nucleus コンポーネントの v2.6.0 以降で利用できます。

Greengrass CLI IPC サービスを使用して、コアデバイスのローカルデプロイおよび Greengrass コンポーネントを管理します。

これらの IPC オペレーションを使用するには、カスタムコンポーネントの依存関係として、バージョン 2.6.0 以降の Greengrass CLI コンポーネントを含めます。その後、カスタムコンポーネントの IPC オペレーションを使用して、次の操作を行うことができます。

- ローカルデプロイを作成し、コアデバイスの Greengrass コンポーネントを変更および設定します。
- コアデバイスの Greengrass コンポーネントを再起動して停止します。
- ローカルデバッグコンソールへのサインインに使用できるパスワードを生成します。

トピック

- 最小 SDK バージョン
- 認証
- CreateLocalDeployment
- ListLocalDeployments
- GetLocalDeploymentStatus
- ListComponents
- GetComponentDetails
- RestartComponent

- StopComponent
- CreateDebugPassword

最小 SDK バージョン

次の表に、AWS IoT Device SDK の最小バージョンを示します。Greengrass CLI IPC サービスとやり取りする際は、これを使用する必要があります。

SDK	最小バージョン	
AWS IoT Device SDK for Java v2	v1.2.10	
AWS IoT Device SDK for Python v2	v1.5.3	
AWS IoT Device SDK for C++ v2	v1.17.0	
AWS IoT Device SDK JavaScript v2 用	v1.12.0	

認証

Greengrass CLI IPC サービスをカスタムコンポーネントで使用するには、コンポーネントがローカルデプロイとコンポーネントを管理できるように承認ポリシーを定義する必要があります。承認ポリシーの定義については、「コンポーネントに IPC オペレーションの実行を許可する」を参照してください。

Greengrass CLI の承認ポリシーには、次のプロパティがあります。

IPC サービス識別子: `aws.greengrass.Cli`

操作	説明	リソース
aws.greengrass#CreateLocalDeployment	コンポーネントがコアデバイスにローカルデプロイを作成できるようにします。	*
aws.greengrass#ListLocalDeployments	コンポーネントがコアデバイスのローカルデプロイを一覧表示できるようにします。	*
aws.greengrass#GetLocalDeploymentStatus	コンポーネントがコアデバイスのローカルデプロイのステータスを取得できるようにします。	すべてのローカルデプロイへのアクセスを許可するローカルデプロイ ID または *。
aws.greengrass#ListComponents	コンポーネントがコアデバイスのコンポーネントを一覧表示できるようにします。	*
aws.greengrass#GetComponentDetails	コンポーネントがコアデバイスのコンポーネントの詳細を取得できるようにします。	すべてのコンポーネントへのアクセスを許可するコンポーネント名 (com.example.HelloWorld など) または *。
aws.greengrass#RestartComponent	コンポーネントがコアデバイスのコンポーネントを再起動できるようにします。	すべてのコンポーネントへのアクセスを許可するコンポーネント名 (com.example.HelloWorld など) または *。
aws.greengrass#StopComponent	コンポーネントがコアデバイスのコンポーネントを停止できるようにします。	すべてのコンポーネントへのアクセスを許可するコンポーネント名 (com.example.HelloWorld など) または *。

操作	説明	リソース
aws.greengrass#CreateDebugPassword	コンポーネントがローカルデバッグコンソールコンポーネントへのサインインに使用するパスワードを生成できるようにします。	*

Example 承認ポリシーの例

次の認証ポリシーの例では、コンポーネントがローカルデプロイを作成し、すべてのローカルデプロイとコンポーネントを表示し、com.example.HelloWorld という名前のコンポーネントを再起動および停止できるようにします。

```
{
  "accessControl": {
    "aws.greengrass.Cli": {
      "com.example.MyLocalManagerComponent:cli:1": {
        "policyDescription": "Allows access to create local deployments and view
 deployments and components.",
        "operations": [
          "aws.greengrass#CreateLocalDeployment",
          "aws.greengrass#ListLocalDeployments",
          "aws.greengrass#GetLocalDeploymentStatus",
          "aws.greengrass#ListComponents",
          "aws.greengrass#GetComponentDetails"
        ],
        "resources": [
          "*"
        ]
      }
    },
    "aws.greengrass.Cli": {
      "com.example.MyLocalManagerComponent:cli:2": {
        "policyDescription": "Allows access to restart and stop the Hello World
 component.",
        "operations": [
          "aws.greengrass#RestartComponent",
          "aws.greengrass#StopComponent"
        ],
        "resources": [
```

```
          "com.example.HelloWorld"
        ]
      }
    }
  }
}
```

CreateLocalDeployment

指定されたコンポーネントレシピ、アーティファクト、ランタイム引数を使用して、ローカルデプロイを作成または更新します。

このオペレーションは、Greengrass CLI のデプロイの create コマンドと同じ機能を提供します。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`recipeDirectoryPath` (Python: `recipe_directory_path`)

(オプション) コンポーネントレシピファイルを含むフォルダへの絶対パス。

`artifactDirectoryPath` (Python: `artifact_directory_path`)

(オプション) デプロイに含めるアーティファクトファイルを含むフォルダへの絶対パス。アーティファクトフォルダには、次のフォルダ構造が含まれている必要があります。

```
/path/to/artifact/folder/component-name/component-version/artifacts
```

`rootComponentVersionsToAdd` (Python: `root_component_versions_to_add`)

(オプション) コアデバイスにインストールするコンポーネントのバージョン。このオブジェクト `ComponentToVersionMap` は、次のキーと値のペアを含むマップです。

`key`

コンポーネントの名前。

`value`

コンポーネントのバージョン。

`rootComponentsToRemove` (Python: `root_components_to_remove`)

(オプション) コアデバイスからアンインストールするコンポーネント。各エントリがコンポーネントの名前となるリストを指定します。

`componentToConfiguration` (Python: `component_to_configuration`)

(オプション) デプロイ内の各コンポーネントの設定更新。このオブジェクト `ComponentToConfiguration` は、次のキーと値のペアを含むマップです。

`key`

コンポーネントの名前。

`value`

コンポーネントの設定更新の JSON オブジェクト。JSON オブジェクトは次の形式である必要があります。

```
{
  "MERGE": {
    "config-key": "config-value"
  },
  "RESET": [
    "path/to/reset/"
  ]
}
```

設定の更新の詳細については、「コンポーネント設定の更新」を参照してください。

`componentToRunWithInfo` (Python: `component_to_run_with_info`)

(オプション) デプロイ内の各コンポーネントのランタイム設定。この設定には、各コンポーネントのプロセスを所有するシステムユーザーと、各コンポーネントに適用するシステム制限が含まれます。このオブジェクト `ComponentToRunWithInfo` は、次のキーと値のペアを含むマップです。

`key`

コンポーネントの名前。

`value`

コンポーネントのランタイム設定。ランタイム設定のパラメータを省略した場合、AWS IoT Greengrass Core ソフトウェアは Greengrass nucleus で設定したデフォルト値を使用します。このオブジェクト (`RunWithInfo`) には、次の情報が含まれます。

`posixUser` (Python: `posix_user`)

(オプション) Linux コアデバイスでこのコンポーネントを実行する際に使用する POSIX システムユーザーおよびオプションのグループ。ユーザーまたはグループを指定する場合は、各 Linux コアデバイス上に存在している必要があります。ユーザーとグループを `user:group` の形式に従ってコロン (`:`) で区切って指定します。グループはオプションです。グループを指定しなかった場合、AWS IoT Greengrass Core ソフトウェアは、ユーザーのプライマリグループを使用します。詳細については、コンポーネントを実行するユーザーを設定する を参照してください。

`windowsUser` (Python: `windows_user`)

(オプション) Windows コアデバイスでこのコンポーネントを実行する際に使用する Windows ユーザー。ユーザーは各 Windows コアデバイスに存在し、その名前とパスワードは LocalSystem アカウントの認証情報マネージャーインスタンスに保存されている必要があります。詳細については、コンポーネントを実行するユーザーを設定する を参照してください。

`systemResourceLimits` (Python: `system_resource_limits`)

(オプション) このコンポーネントのプロセスに適用されるシステムリソースの制限。システムリソースの制限を、ジェネリックおよびコンテナ化されていない Lambda コンポーネントプロセスに適用することができます。詳細については、コンポーネントのシステムリソース制限を設定する を参照してください。

AWS IoT Greengrass は、現在 Windows コアデバイスにこの機能をサポートしていません。

このオブジェクト (`SystemResourceLimits`) には、次の情報が含まれます。

`cpus`

(オプション) このコンポーネントのプロセスがコアデバイスで使用できる CPU 時間の最大量。コアデバイスの合計 CPU 時間は、デバイスの CPU コア数と同じです。例えば、4 つの CPU コアを持つコアデバイスの場合は、この値を `2` に設定することで、このコンポーネントのプロセスを各 CPU コアの 50% の使用率に制限することができます。CPU コアが 1 つのデバイスの場合は、この値を `0.25` に設定することで、このコンポーネントのプロセスを CPU の 25% の使用率に制限することができます。この値を CPU コア数よりも大きい値に設定すると、AWS IoT Greengrass Core ソフトウェアは、コンポーネントの CPU 使用率に制限をかけません。

`memory`

(オプション) このコンポーネントのプロセスがコアデバイスで使用できる RAM の最大量 (キロバイト単位)。

`groupName` (Python: `group_name`)

(オプション) このデプロイでターゲットにするモノのグループの名前。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`deploymentId` (Python: `deployment_id`)

リクエストによって作成されたローカルデプロイの ID。

ListLocalDeployments

過去 10 回分のローカルデプロイのステータスを取得します。

このオペレーションは、Greengrass CLI のデプロイの list コマンドと同じ機能を提供します。

リクエスト

このオペレーションのリクエストはパラメータを持ちません。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`localDeployments` (Python: `local_deployments`)

ローカルデプロイのリスト。このリストの各オブジェクトは、次の情報が含まれる `LocalDeployment` オブジェクトです。

`deploymentId` (Python: `deployment_id`)

ローカルデプロイの ID。

`status`

ローカルデプロイのステータス。この列挙型 (`DeploymentStatus`) には以下の値があります。

- QUEUED
- IN_PROGRESS
- SUCCEEDED
- FAILED

GetLocalDeploymentStatus

ローカルデプロイのステータスを取得します。

このオペレーションは、Greengrass CLI のデプロイの status コマンドと同じ機能を提供します。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`deploymentId` (Python: `deployment_id`)

取得するローカルデプロイの ID。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`deployment`

ローカルデプロイ。このオブジェクト (`LocalDeployment`) には、次の情報が含まれます。

`deploymentId` (Python: `deployment_id`)

ローカルデプロイの ID。

`status`

ローカルデプロイのステータス。この列挙型 (`DeploymentStatus`) には以下の値があります。

- QUEUED
- IN_PROGRESS
- SUCCEEDED

- FAILED

ListComponents

コアデバイスの各ルートコンポーネントの名前、バージョン、ステータス、および設定を取得します。ルートコンポーネントは、デプロイ時に指定するコンポーネントです。このレスポンスには、他のコンポーネントとの依存関係の中でインストールされるコンポーネントは含まれません。

このオペレーションは、Greengrass CLI のコンポーネントの list コマンドと同じ機能を提供します。

リクエスト

このオペレーションのリクエストはパラメータを持ちません。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`components`

コアデバイスのルートコンポーネントのリスト。このリストの各オブジェクトは、次の情報が含まれる `ComponentDetails` オブジェクトです。

`componentName` (Python: `component_name`)

コンポーネントの名前。

`version`

コンポーネントのバージョン。

`state`

コンポーネントの状態。これは、次のいずれかの状態になります。

- BROKEN
- ERRORED
- FINISHED
- INSTALLED
- NEW

- RUNNING
- STARTING
- STOPPING

configuration

JSON オブジェクトとしてのコンポーネントの設定。

GetComponentDetails

コアデバイスのコンポーネントのバージョン、ステータス、および設定を取得します。

このオペレーションは、Greengrass CLI のコンポーネントの details コマンドと同じ機能を提供します。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

componentName (Python: component_name)

取得するコンポーネントの名前。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

componentDetails (Python: component_details)

コンポーネントの詳細。このオブジェクト (ComponentDetails) には、次の情報が含まれます。

componentName (Python: component_name)

コンポーネントの名前。

version

コンポーネントのバージョン。

state

コンポーネントの状態。これは、次のいずれかの状態になります。

- BROKEN
- ERRORED
- FINISHED
- INSTALLED
- NEW
- RUNNING
- STARTING
- STOPPING

configuration

JSON オブジェクトとしてのコンポーネントの設定。

RestartComponent

コアデバイスのコンポーネントを再起動します。

> **Note**
>
> どのコンポーネントでも再起動できますが、ジェネリックコンポーネントのみを再起動することをお勧めします。

このオペレーションは、Greengrass CLI のコンポーネントの restart コマンドと同じ機能を提供します。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

componentName (Python: component_name)

コンポーネントの名前。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`restartStatus` (Python: `restart_status`)

再起動リクエストのステータス。リクエストのステータスは次のいずれかになります。

- `SUCCEEDED`
- `FAILED`

`message`

リクエストが失敗した際の、コンポーネントが再起動に失敗した理由に関するメッセージ。

StopComponent

コアデバイスのコンポーネントのプロセスを停止します。

> **Note**
> どのコンポーネントも停止できますが、ジェネリックコンポーネントのみを停止することをお勧めします。

このオペレーションは、Greengrass CLI のコンポーネントの stop コマンドと同じ機能を提供します。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`componentName` (Python: `component_name`)

コンポーネントの名前。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`stopStatus` (Python: `stop_status`)

停止リクエストのステータス。リクエストのステータスは次のいずれかになります。

- `SUCCEEDED`

- `FAILED`

`message`

リクエストが失敗した際の、コンポーネントが停止に失敗した理由に関するメッセージ。

CreateDebugPassword

ローカルデバッグコンソールコンポーネントへのサインインに使用できる、ランダムなパスワードを生成します。パスワードは、生成されてから 8 時間後に期限切れになります。

このオペレーションは、Greengrass CLI の get-debug-password コマンドと同じ機能を提供します。

リクエスト

このオペレーションのリクエストはパラメータを持ちません。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`username`

サインインに使用するユーザー名。

`password`

サインインに使用するパスワード。

`passwordExpiration` (Python: `password_expiration`)

パスワードの有効期限が切れる時刻。

`certificateSHA256Hash` (Python: `certificate_sha256_hash`)

HTTPS が有効な場合にローカルデバッグコンソールが使用する自己署名証明書の SHA-256 フィンガープリント。ローカルデバッグコンソールを開く際に、このフィンガープリントを使用して、証明書が正当であり接続が安全であることを確認します。

`certificateSHA1Hash` (Python: `certificate_sha1_hash`)

HTTPS が有効な場合にローカルデバッグコンソールが使用する自己署名証明書の SHA-1 フィンガープリント。ローカルデバッグコンソールを開く際に、このフィンガープリントを使用して、証明書が正当であり接続が安全であることを確認します。

クライアントデバイスを認証して承認する

> **Note**
>
> この機能は、Greengrass nucleus コンポーネントの v2.6.0 以降で利用できます。

クライアントデバイス認証 IPC サービスを使用して、クライアントデバイスなどのローカル IoT デバイスが接続できるカスタムのローカルブローカーコンポーネントを開発します。

これらの IPC オペレーションを使用するには、カスタムコンポーネントの依存関係として、バージョン 2.2.0 以降のクライアントデバイス認証コンポーネントを含めます。その後、カスタムコンポーネントの IPC オペレーションを使用して、次の操作を行うことができます。

- コアデバイスに接続するクライアントデバイスの ID の検証。
- クライアントデバイスがコアデバイスに接続するためのセッションの作成。
- クライアントデバイスにアクションを実行するアクセス許可があるかどうかの検証。
- コアデバイスのサーバー証明書が更新された際の通知の受け取り。

トピック

最小 SDK バージョン

次の表に、AWS IoT Device SDK の最小バージョンを示します。クライアントデバイス認証 IPC サービスとやり取りする際は、これを使用する必要があります。

SDK	最小バージョン	
AWS IoT Device SDK for Java v2	v1.9.3	
AWS IoT Device SDK for Python v2	v1.11.3	
AWS IoT Device SDK for C++ v2	v1.18.3	
AWS IoT Device SDK JavaScript v2 用	v1.12.0	

認証

カスタムコンポーネントでクライアントデバイス認証 IPC サービスを使用するには、コンポーネントがこれらのオペレーションを実行できるように承認ポリシーを定義する必要があります。承認ポリシーの定義については、「コンポーネントに IPC オペレーションの実行を許可する」を参照してください。

クライアントデバイスの認証および承認の承認ポリシーには、次のプロパティがあります。

IPC サービス識別子: `aws.greengrass.clientdevices.Auth`

操作	説明	リソース
`aws.greengrass#VerifyClientDeviceIdentity`	コンポーネントがクライアントデバイスの ID を検証できるようにします。	*
`aws.greengrass#GetClientDeviceAuthToken`	コンポーネントがクライアントデバイスの認証情報を検証し、そのクライアントデバイスのセッションを作成できるようにします。	*

操作	説明	リソース
aws.greengrass#AuthorizeClientDeviceAction	クライアントデバイスにアクションを実行するアクセス許可があるかどうかを、コンポーネントが検証できるようにします。	*
aws.greengrass#SubscribeToCertificateUpdates	コアデバイスのサーバー証明書が更新されたときに、コンポーネントが通知を受け取ることができるようにします。	*
*	コンポーネントがすべてのクライアントデバイス認証 IPC サービスのオペレーションを実行できるようにします。	*

承認ポリシーの例

次の承認ポリシーの例を参照して、コンポーネントの承認ポリシーの設定に役立てることができます。

Example 承認ポリシーの例

次の承認ポリシーの例では、コンポーネントがすべてのクライアントデバイス認証の IPC オペレーションを実行できるようにします。

```
{
  "accessControl": {
    "aws.greengrass.clientdevices.Auth": {
      "com.example.MyLocalBrokerComponent:clientdevices:1": {
        "policyDescription": "Allows access to authenticate and authorize client
 devices.",
        "operations": [
          "aws.greengrass#VerifyClientDeviceIdentity",
          "aws.greengrass#GetClientDeviceAuthToken",
          "aws.greengrass#AuthorizeClientDeviceAction",
          "aws.greengrass#SubscribeToCertificateUpdates"
```

```
        ],
        "resources": [
          "*"
        ]
      }
    }
  }
}
```

VerifyClientDeviceIdentity

クライアントデバイスの ID の検証。このオペレーションは、クライアントデバイスが有効な AWS IoT のモノであるかどうかを検証します。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`credential`

クライアントデバイスの認証情報。このオブジェクト (`ClientDeviceCredential`) には、次の情報が含まれます。

`clientDeviceCertificate` (Python: `client_device_certificate`)

クライアントデバイスの X.509 デバイス証明書。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`isValidClientDevice` (Python: `is_valid_client_device`)

クライアントデバイスの ID が有効かどうか。

GetClientDeviceAuthToken

クライアントデバイスの認証情報を検証し、クライアントデバイスのセッションを作成します。このオペレーションは、クライアントデバイスのアクションを承認するために、後続のリクエストで使用できるセッショントークンを返します。

クライアントデバイスを正常に接続するには、クライアントデバイス認証コンポーネントで、クライアントデバイスが使用するクライアント ID の `mqtt:connect` アクセス許可が付与される必要があります。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`credential`

クライアントデバイスの認証情報。このオブジェクト (`CredentialDocument`) には、次の情報が含まれます。

`mqttCredential` (Python: `mqtt_credential`)

クライアントデバイスの MQTT 認証情報。クライアントデバイスが接続に使用するクライアント ID と証明書を指定します。このオブジェクト (`MQTTCredential`) には、次の情報が含まれます。

`clientId` (Python: `client_id`)

接続に使用するクライアント ID。

`certificatePem` (Python: `certificate_pem`)

接続に使用する X.509 デバイス証明書。

`username`

> **Note**
> このプロパティは現在使用されていません。

`password`

> **Note**
> このプロパティは現在使用されていません。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`clientDeviceAuthToken` (Python: `client_device_auth_token`)

クライアントデバイスのセッショントークン。このセッショントークンを後続のリクエストで使用して、このクライアントデバイスのアクションを承認できます。

AuthorizeClientDeviceAction

クライアントデバイスに、リソースに対するアクションを実行するアクセス許可があるかどうかを検証します。クライアントデバイスの承認ポリシーは、コアデバイスに接続している際にクライアントデバイスが実行できるアクセス許可を指定します。クライアントデバイスの承認ポリシーは、クライアントデバイス認証コンポーネントを設定する際に定義します。

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`clientDeviceAuthToken` (Python: `client_device_auth_token`)

クライアントデバイスのセッショントークン。

`operation`

承認するオペレーション。

`resource`

クライアントデバイスがオペレーションを実行するリソース。

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`isAuthorized` (Python: `is_authorized`)

クライアントデバイスがリソースに対するオペレーションの実行を許可されているかどうか。

SubscribeToCertificateUpdates

コアデバイスの新しいサーバー証明書が更新されるたびにそれを受け取るようにサブスクライブします。サーバー証明書が更新されると、新しいサーバー証明書を使用してブローカーをリロードする必要があります。

クライアントデバイス認証コンポーネントは、デフォルトでは 7 日ごとにサーバー証明書を更新します。更新の間隔は、2 日から 10 日の間で設定できます。

このオペレーションはサブスクリプションオペレーションで、イベントメッセージのストリームをサブスクライブするというものです。このオペレーションを使用するには、イベントメッセージ、エラー、およびストリームクロージャを処理する関数を使用して、ストリームレスポンスハンドラーを定義します。(詳細については、IPC イベントストリームへのサブスクライブ を参照してください)。

イベントメッセージの種類: `CertificateUpdateEvent`

リクエスト

このオペレーションのリクエストには以下のパラメータがあります。

`certificateOptions` (Python: `certificate_options`)

サブスクライブする証明書更新の種類。このオブジェクト (`CertificateOptions`) には、次の情報が含まれます。

`certificateType` (Python: `certificate_type`)

サブスクライブする証明書更新の種類。次のオプションを選択します。

- `SERVER`

レスポンス

このオペレーションのレスポンスには以下の情報が含まれます。

`messages`

メッセージのストリーム。このオブジェクト (`CertificateUpdateEvent`) には、次の情報が含まれます。

`certificateUpdate` (Python: `certificate_update`)

新しい証明書に関する情報。このオブジェクト (`CertificateUpdate`) には、次の情報が含まれます。

`certificate`

証明書。

`privateKey` (Python: `private_key`)

証明書のプライベートキー。

`publicKey` (Python: `public_key`)

証明書のパブリックキー。

`caCertificates` (Python: `ca_certificates`)

証明書の CA 証明書チェーン内の認証局 (CA) 証明書のリスト。

ローカル IoT デバイスとやり取りする

「クライアントデバイス」とは、MQTT 経由で Greengrass コアデバイスと接続および通信を行うローカル IoT デバイスです。クライアントデバイスをコアデバイスに接続すると、次の操作を実行できます。

- Greengrass コンポーネントの MQTT メッセージとやり取りする。
- クライアントデバイスと AWS IoT Core の間でメッセージやデータをリレーする。
- Greengrass コンポーネント内のクライアントデバイスシャドウとやり取りします。
- クライアントデバイスシャドウを AWS IoT Core と同期させる。

コアデバイスに接続する際、クライアントデバイスは「クラウド検出」を使用できます。クライアントデバイスは AWS IoT Greengrass クラウドサービスに接続して、接続可能なコアデバイスに関する情報を取得します。その後、コアデバイスに接続してメッセージを処理し、AWS IoT Core クラウドサービスにデータを同期させます。

コアデバイスを AWS IoT モノに接続して通信するための設定方法を説明するチュートリアルに沿って設定を行えます。このチュートリアルでは、クライアントデバイスを操作するカスタム Greengrass コンポーネントの開発方法についても説明します。詳細については、「チュートリアル: MQTT 経由でローカル IoT デバイスとやり取りする」を参照してください。

トピック

AWS から提供されるクライアントデバイスコンポーネント

AWS IoT Greengrass は、コアデバイスにデプロイ可能な次のパブリックコンポーネントを提供します。これらのコンポーネントを使用すると、クライアントデバイスはコアデバイスに接続して通信できるようになります。

> **Note**
>
> AWS が提供する複数のコンポーネントは、Greengrass nucleus の特定マイナーバージョンに依存します。この従属関係により、Greengrass nucleus を新しいマイナーバージョンに更新するとき、これらのコンポーネントを更新する必要があります。各コンポーネントが依存する nucleus の特定バージョンの情報については、対応するコンポーネントのトピックを参照してください。nucleus の更新の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネントのコンポーネントタイプが汎用と Lambda の両方である場合、コンポーネントの現在のバージョンは汎用タイプであり、コンポーネントの以前のバージョンは Lambda タイプです。

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
クライアントデバイス認証	クライアントデバイスと呼ばれるローカル IoT デバイスがコアデバイスに接続できるようにします。	プラグイン	Linux、Windows	はい
IP ディテクター	MQTT ブローカーの接続情報を AWS IoT Greengrass に報告し、クライアントデバイスが接続方法を検出できるようにします。	プラグイン	Linux、Windows	はい

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
MQTT ブリッジ	クライアントデバイス、ローカル AWS IoT Greengrass パブリッシュ/サブスクライブ、AWS IoT Core の間で MQTT メッセージをリレーします。	プラグイン	Linux、Windows	はい
MQTT 3.1.1 ブローカー (モケット)	クライアントデバイスとコアデバイスの間のメッセージを処理する、MQTT 3.1.1 ブローカーを実行します。	プラグイン	Linux、Windows	はい
MQTT 5 ブローカー (EMQX)	クライアントデバイスとコアデバイスの間のメッセージを処理する、MQTT 5 ブローカーを実行します。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
シャドウマネージャー	コアデバイス上のシャドウとの対話を有効にします。AWS IoT デバイスシャドウサービスで、シャドウドキュメントのストレージとローカルシャドウ状態の同期も管理します。	プラグイン	Linux、Windows	はい

クライアントデバイスをコアデバイスに接続する

クラウドディスカバリを設定して、クライアントデバイスをコアデバイスに接続することができます。クラウドディスカバリを設定すると、クライアントデバイスは AWS IoT Greengrass クラウドサービスに接続して、接続可能なコアデバイスに関する情報を取得することができます。その後、クライアントデバイスは、正常に接続されるまで、各コアデバイスへの接続を試みることができます。

クラウドディスカバリを使用するには、以下の操作を行う必要があります。

- クライアントデバイスを、接続できるコアデバイスに関連付けます。
- クライアントデバイスが各コアデバイスに接続できる MQTT ブローカエンドポイントを指定します。
- クライアントデバイスへのサポートを可能にするコンポーネントをコアデバイスにデプロイします。

  オプションのコンポーネントをデプロイして、次の操作を行うこともできます。

  - クライアントデバイス、Greengrass コンポーネント、AWS IoT Core クラウドサービスの間でメッセージをリレーします。

- コアデバイスの MQTT ブローカーエンドポイントを自動で管理します。
- ローカルクライアントのデバイスシャドウを管理し、AWS IoT Core のクラウドサービスとシャドウを同期します。

また、コアデバイスの AWS IoT ポリシーを見直して更新し、クライアントデバイスに接続するために必要な権限があることを確認する必要があります。詳細については、「要件」を参照してください。

クラウドディスカバリを設定したら、クライアントデバイスとコアデバイス間の通信をテストすることができます。詳細については、「クライアントデバイス通信をテストする」を参照してください。

トピック

要件

クライアントデバイスをコアデバイスに接続するには、以下を使用する必要があります。

- コアデバイスで、Greengrass nucleus v2.2.0 以降が実行されている必要があります
- コアデバイスが動作している AWS リージョン内にある、AWS アカウント の AWS IoT Greengrass に関連付けられたGreengrass サービスロール。詳細については、「Greengrass サービスロールを設定する」を参照してください。
- コアデバイスの AWS IoT ポリシーでは以下のアクセス許可を付与する必要があります。

- `greengrass:PutCertificateAuthorities`
- `greengrass:VerifyClientDeviceIdentity`
- `greengrass:VerifyClientDeviceIoTCertificateAssociation`
- `greengrass:GetConnectivityInfo`
- `greengrass:UpdateConnectivityInfo` - (オプション) このアクセス許可は、コアデバイスのネットワーク接続に関する情報を AWS IoT Greengrass クラウドサービスに報告する IP 検出コンポーネントを使用するために必要です。
- `iot:GetThingShadow`、`iot:UpdateThingShadow`、および `iot:DeleteThingShadow` – (オプション) これらのアクセス許可は、クライアントデバイスシャドウと AWS IoT Core を同期する、シャドウマネージャーコンポーネントを使用するために必要です。この機能を使用するには、Greengrass nucleus v2.6.0 以降、シャドウマネージャー v2.2.0 以降、および MQTT ブリッジ v2.2.0 以降が必要です。

詳細については、「AWS IoTモノポリシーを設定する」を参照してください。

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアのインストール時にデフォルトの AWS IoT ポリシーを使用した場合、コアデバイスにはすべての AWS IoT Greengrass アクション (`greengrass:*`) へのアクセスを許可する AWS IoT ポリシーが設定されています。

- クライアントデバイスとして接続できる AWS IoT モノ。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT リソースを作成する」を参照してください。
- クライアントデバイスは、クライアント ID を使用して接続する必要があります。クライアント ID はモノの名前です。他のクライアント ID は受け付けられません。
- クライアントデバイスの AWS IoT ポリシーでは、`greengrass:Discover` アクセス許可を付与する必要があります。詳細については、「クライアントデバイス向けの最低限の AWS IoT ポリシー」を参照してください。

トピック

- Greengrass サービスロールを設定する
- AWS IoTモノポリシーを設定する

Greengrass サービスロールを設定する

Greengrass サービスロールは、ユーザーに代わって AWS サービスからリソースへのアクセスを AWS IoT Greengrass に許可する AWS Identity and Access Management (IAM) サービスロールです。このロールにより、AWS IoT Greengrass でクライアントデバイスの ID を確認し、コアデバイスの接続情報を管理できるようになります。

このリージョンでまだ Greengrass サービスロールを設定していない場合は、このリージョンの AWS アカウント の AWS IoT Greengrass に Greengrass サービスロールに関連付ける必要があります。

AWS IoT Greengrass コンソールで [Configure core device discovery] (コアデバイスディスカバリを設定する) ページを使用すると、AWS IoT Greengrass が Greengrass サービスロールを設定します。そうでない場合は、AWS IoT コンソールや AWS IoT Greengrass APIを使って、手動で設定することができます。

このセクションでは、Greengrass サービスロールが設定されているかどうかを確認します。設定されていない場合は、このリージョンの AWS アカウント の AWS IoT Greengrass のために関連付ける新しい Greengrass サービスロールを作成します。

Greengrass サービスロールを設定する (コンソール)

1. Greengrass サービスロールが、このリージョンの AWS アカウント の AWS IoT Greengrass に関連付けられていることを確認します。以下の操作を実行します。

   a. AWS IoT コンソールに移動します。
   b. ナビゲーションペインで [設定] を選択します。
   c. [Greengrass service role] (Greengrass サービスロール) セクションで、[Current service role] (現在のサービスロール) をクリックし、Greengrass サービスロールが関連付けられているかどうかを確認します。

      Greengrass サービスロールが関連付けられている場合、IP ディテクターコンポーネントを使用する要件を満たしています。「AWS IoTモノポリシーを設定する」へ進んでください。
2. Greengrass サービスロールがこのリージョンの AWS アカウント の AWS IoT Greengrass に関連付けられていない場合は、Greengrass サービスロールを作成して関連付けます。以下の操作を実行します。

   a. [IAM console] (IAM コンソール) に入ります。
   b. [Roles (ロール)] を選択します。

c. [Create role] (ロールの作成) を選択します。

d. [Create role] (ロールの作成) ページで、次の手順を実行します。

   i. [Trusted entity type] (信頼できるエンティティタイプ) で、[AWS のサービス] を選択します。

   ii. [ユースケース] の下にある [その他の AWS のサービス のユースケース] で、[Greengrass] を選択し、さらに [Greengrass] を選択します。このオプションは、このロールを継承することができる信頼できるエンティティとして AWS IoT Greengrass を追加することを指定します。

   iii. [次へ] をクリックします。

   iv. [Permissions policies] (アクセス許可ポリシー) で、AWSGreengrassResourceAccessRolePolicy を選択し、ロールにアタッチします。

   v. [次へ] をクリックします。

   vi. [Role name] (ロール名) に、このロールの名前 (**`Greengrass_ServiceRole`** など) を入力します。

   vii. [ロールの作成] を選択します。

e. AWS IoT コンソールに移動します。

f. ナビゲーションペインで [設定] を選択します。

g. 左[Greengrass service role] (Greengrass サービスロール) セクションで、[Attach role] (ロールを添付する) を選択します。

h. [Update Greengrass service role] (Greengrass サービスロールを更新する) モーダルで作成した IAM ロールを選択し、[Attach role] (ロールを割り当てる) を選択します。

Greengrass サービスロールを設定する (AWS CLI)

1. Greengrass サービスロールが、このリージョンの AWS アカウント の AWS IoT Greengrass に関連付けられていることを確認します。

   ```
   aws greengrassv2 get-service-role-for-account
   ```

   Greengrass サービスロールが関連付けられている場合、この動作により、ロールに関する情報が含まれたレスポンスが返されます。

   Greengrass サービスロールが関連付けられている場合、IP ディテクターコンポーネントを使用する要件を満たしています。「AWS IoTモノポリシーを設定する」へ進んでください。

2. Greengrass サービスロールがこのリージョンの AWS アカウント の AWS IoT Greengrass に関連付けられていない場合は、Greengrass サービスロールを作成して関連付けます。以下の操作を実行します。

   a. AWS IoT Greengrass がロールを継承できるように許可する信頼ポリシーを持つロールを作成します。この例では、`Greengrass_ServiceRole` という名前のロールを作成しますが、別の名前を使用できます。また、信頼ポリシーには、`aws:SourceArn` および `aws:SourceAccount` グローバル条件コンテキストキーも含めて、混乱した代理によるセキュリティ問題を防止することをお勧めします。条件コンテキストキーを使用すると、指定したアカウントと Greengrass ワークスペースからのリクエストのみを許可するようにアクセスを制限できます。混乱した代理に関する問題の詳細については、「サービス間の混乱した代理の防止」を参照してください。

      Linux or Unix

      ```
      aws iam create-role --role-name Greengrass_ServiceRole --assume-role-policy-
      document '{
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Principal": {
              "Service": "greengrass.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
              "ArnLike": {
                "aws:SourceArn": "arn:aws:greengrass:region:account-id:*"
              },
              "StringEquals": {
                "aws:SourceAccount": "account-id"
              }
            }
          }
        ]
      }'
      ```

      Windows Command Prompt (CMD)

      ```
      aws iam create-role --role-name Greengrass_ServiceRole --assume-role-policy-
      document "{\\"Version\\":\\"2012-10-17\\",\\"Statement\\":[{\\"Effect\
      ```

```
\":\\"Allow\\",\\"Principal\\":{\\"Service\\":\\"greengrass.amazonaws.com
\\"},\\"Action\\":\\"sts:AssumeRole\\",\\"Condition\\":{\\"ArnLike\\":
{\\"aws:SourceArn\\":\\"arn:aws:greengrass:region:account-id:*\\"},\
\"StringEquals\\":{\\"aws:SourceAccount\\":\\"account-id\\"}}}]}"
```

PowerShell

```
aws iam create-role --role-name Greengrass_ServiceRole --assume-role-policy-
document '{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "greengrass.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnLike": {
          "aws:SourceArn": "arn:aws:greengrass:region:account-id:*"
        },
        "StringEquals": {
          "aws:SourceAccount": "account-id"
        }
      }
    }
  ]
}'
```

b. 出力のロールメタデータからロールの ARN をコピーします。ARN を使用して、ロールをアカウントに関連付けます。

c. AWSGreengrassResourceAccessRolePolicy ポリシーをロールにアタッチします。

```
aws iam attach-role-policy --role-name Greengrass_ServiceRole --policy-arn
 arn:aws:iam::aws:policy/service-role/AWSGreengrassResourceAccessRolePolicy
```

d. Greengrass サービスロールを AWS アカウント の AWS IoT Greengrass に関連付けます。*role-arn* をサービスロールの ARN に置き換えます。

```
aws greengrassv2 associate-service-role-to-account --role-arn role-arn
```

成功すると、次のレスポンスが返されます。

```
{
  "associatedAt": "timestamp"
}
```

AWS IoTモノポリシーを設定する

コアデバイスは X.509 デバイス証明書を使用して、AWS への接続を許可します。デバイス証明書に AWS IoT ポリシーをアタッチして、コアデバイスのアクセス許可を定義します。詳細については、「データプレーンオペレーションの AWS IoT ポリシー と クライアントデバイスをサポートするための最低限の AWS IoT ポリシー」を参照してください。

クライアントデバイスをコアデバイスに接続するには、コアデバイスの AWS IoT ポリシーで、次のアクセス許可を許可する必要があります。

- `greengrass:PutCertificateAuthorities`
- `greengrass:VerifyClientDeviceIdentity`
- `greengrass:VerifyClientDeviceIoTCertificateAssociation`
- `greengrass:GetConnectivityInfo`
- `greengrass:UpdateConnectivityInfo` - (オプション) このアクセス許可は、コアデバイスのネットワーク接続に関する情報を AWS IoT Greengrass クラウドサービスに報告する IP 検出コンポーネントを使用するために必要です。
- `iot:GetThingShadow`、`iot:UpdateThingShadow`、および `iot:DeleteThingShadow` – (オプション) これらのアクセス許可は、クライアントデバイスシャドウと AWS IoT Core を同期する、シャドウマネージャーコンポーネントを使用するために必要です。この機能を使用するには、Greengrass nucleus v2.6.0 以降、シャドウマネージャー v2.2.0 以降、および MQTT ブリッジ v2.2.0 以降が必要です。

このセクションでは、コアデバイスの AWS IoT ポリシーを確認し、必要なアクセス許可が不足している場合には追加します。AWS IoT Greengrass Core ソフトウェアインストーラを使用してリソースをプロビジョニングした場合、コアデバイスには、すべての AWS IoT Greengrass アクション (`greengrass:*`) へのアクセスを許可する AWS IoT ポリシーが存在します。この場合に、デバイスシャドウを AWS IoT Core と同期するためにシャドウマネージャーコンポーネントをデプロイする

予定ならば、AWS IoT ポリシーを更新する必要があります。それ以外の場合、このセクションはスキップできます。

AWS IoT モノポリシーを設定する (コンソール)

1. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Core devices] (コアデバイス) を選択します。
2. [Core devices] (コアデバイス) ページで、更新するコアデバイスを選択します。
3. コアデバイスの詳細ページで、コアデバイスの [Thing] (モノ) へのリンクを選択します。このリンクをクリックすると、AWS IoT コンソールの [thing details] (モノ詳細) ページが開きます。
4. [thing details] (モノ詳細) ページで、[Certificates] (証明書) を選択します。
5. [Certificates] (証明書) タブで、モノのアクティブな証明書を選択します。
6. [certificate details] (証明書詳細) ページで、[Policies] (ポリシー) を選択します。
7. [Policies] (ポリシー) タブで、確認して更新する AWS IoT ポリシーを選択します。コアデバイスのアクティブな証明書にアタッチされている任意のポリシーに、必要なアクセス許可を追加できます。

   > **Note**
   >
   > リソースのプロビジョニングに AWS IoT Greengrass Core ソフトウェアインストーラを使用している場合、2 つの AWS IoT ポリシーが存在します。存在する場合は、GreengrassV2IoTThingPolicy という名前のポリシーを選択することをお勧めします。クイックインストーラで作成するコアデバイスは、デフォルトでこのポリシー名を使用します。このポリシーにアクセス許可を追加すると、このポリシーを使用する他のコアデバイスにもこれらのアクセス許可が付与されます。

8. [policy overview] (ポリシーの概要) で、[Edit active version] (アクティブなバージョンの編集) を選択します。
9. 必要なアクセス許可についてポリシーを確認し、不足していれば必要なアクセス許可を追加します。

   - `greengrass:PutCertificateAuthorities`
   - `greengrass:VerifyClientDeviceIdentity`
   - `greengrass:VerifyClientDeviceIoTCertificateAssociation`
   - `greengrass:GetConnectivityInfo`

- `greengrass:UpdateConnectivityInfo` - (オプション) このアクセス許可は、コアデバイスのネットワーク接続に関する情報を AWS IoT Greengrass クラウドサービスに報告する IP 検出コンポーネントを使用するために必要です。
- `iot:GetThingShadow`、`iot:UpdateThingShadow`、および `iot:DeleteThingShadow` – (オプション) これらのアクセス許可は、クライアントデバイスシャドウと AWS IoT Core を同期する、シャドウマネージャーコンポーネントを使用するために必要です。この機能を使用するには、Greengrass nucleus v2.6.0 以降、シャドウマネージャー v2.2.0 以降、および MQTT ブリッジ v2.2.0 以降が必要です。

10. (オプション) コアデバイスがシャドウを AWS IoT Core と同期できるようにするには、ポリシーに次のステートメントを追加します。クライアントのデバイスシャドウとやり取りは行うものの、それらを AWS IoT Core と同期させない場合には、この手順をスキップしてください。*region* および *account-id* は、使用するリージョンと、自分の AWS アカウント 番号に置き換えます。

    - このサンプルステートメントは、すべてのモノのデバイスシャドウへのアクセスを許可します。セキュリティのベストプラクティスに従うには、コアデバイスと、コアデバイスに接続するクライアントデバイスのみにアクセスを制限します。詳細については、「クライアントデバイスをサポートするための最低限の AWS IoT ポリシー」を参照してください。

    ```
    {
      "Effect": "Allow",
      "Action": [
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:DeleteThingShadow"
      ],
      "Resource": [
        "arn:aws:iot:region:account-id:thing/*"
      ]
    }
    ```

    このステートメントを追加した後のポリシードキュメントは、次の例のようになります。

    ```
    {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
    ```

```
      "Action": [
        "iot:Connect",
        "iot:Publish",
        "iot:Subscribe",
        "iot:Receive",
        "greengrass:*"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:DeleteThingShadow"
      ],
      "Resource": [
        "arn:aws:iot:region:account-id:thing/*"
      ]
    }
  ]
}
```

11. 新しいポリシーバージョンをアクティブなバージョンとして設定するには、[Policy version status] (ポリシーバージョンのステータス) で、[Set the edited version as the active version for this policy] (編集したバージョンをこのポリシーのアクティブバージョンとして設定) を選択します。
12. [Save as new version] (新しいバージョンとして保存) を選択します。

AWS IoT モノのポリシーを設定する (AWS CLI)

1. コアデバイスの AWS IoT モノのプリンシパルをリスト表示します。モノのプリンシパルは X.509 デバイス証明書またはその他の識別子にすることができます。次のコマンドを実行し、をコアデバイスの名前*MyGreengrassCore*に置き換えます。

   ```
   aws iot list-thing-principals --thing-name MyGreengrassCore
   ```

   この動作は、コアデバイスのモノのプリンシパルをリスト表示するレスポンスを返します。

   ```
   {
       "principals": [
   ```

```
        "arn:aws:iot:us-west-2:123456789012:cert/certificateId"
    ]
}
```

2. コアデバイスのアクティブな証明書を特定します。以下のコマンドを実行して、アクティブな証明書が見つかるまで、*certificateId* を前の手順からの各証明書の ID に置き換えます。証明書 ID は、証明書 ARN の末尾にある 16 進数の文字列です。--query 引数は、証明書のステータスのみを出力するように指定しています。

```
aws iot describe-certificate --certificate-id certificateId --query
 'certificateDescription.status'
```

この操作では、証明書の状態が文字列で返されます。例えば、証明書がアクティブな場合、この操作は "ACTIVE" を出力します。

3. 証明書にアタッチされている AWS IoT ポリシーをリスト表示します。次のコマンドを実行し、証明書 ARN を証明書の ARN に置き換えます。

```
aws iot list-principal-policies --principal arn:aws:iot:us-
west-2:123456789012:cert/certificateId
```

この操作は、証明書にアタッチされている AWS IoT ポリシーのリストを返します。

```
{
    "policies": [
        {
            "policyName":
 "GreengrassTESCertificatePolicyMyGreengrassCoreTokenExchangeRoleAlias",
            "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassTESCertificatePolicyMyGreengrassCoreTokenExchangeRoleAlias"
        },
        {
            "policyName": "GreengrassV2IoTThingPolicy",
            "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy"
        }
    ]
}
```

4. 表示して更新するポリシーを選択します。

> **Note**
>
> リソースのプロビジョニングに AWS IoT Greengrass Core ソフトウェアインストーラを使用している場合、2 つの AWS IoT ポリシーが存在します。存在する場合は、GreengrassV2IoTThingPolicy という名前のポリシーを選択することをお勧めします。クイックインストーラで作成するコアデバイスは、デフォルトでこのポリシー名を使用します。このポリシーにアクセス許可を追加すると、このポリシーを使用する他のコアデバイスにもこれらのアクセス許可が付与されます。

5. ポリシーのドキュメントを取得します。次のコマンドを実行し、*GreengrassV2IoTThingPolicy* をポリシーの名前に置き換えます。

```
aws iot get-policy --policy-name GreengrassV2IoTThingPolicy
```

この操作は、ポリシーのドキュメントとポリシーに関するその他の情報が含まれるレスポンスを返します。ポリシードキュメントは、文字列としてシリアル化された JSON オブジェクトです。

```
{
    "policyName": "GreengrassV2IoTThingPolicy",
    "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy",
    "policyDocument": "{\
  \\"Version\\": \\"2012-10-17\\",\
  \\"Statement\\": [\
    {\
      \\"Effect\\": \\"Allow\\",\
      \\"Action\\": [\
                \\"iot:Connect\\",\
                \\"iot:Publish\\",\
                \\"iot:Subscribe\\",\
                \\"iot:Receive\\",\
                \\"greengrass:*\\"\
],\
      \\"Resource\\": \\"*\\"\
    }\
  ]\
}",
    "defaultVersionId": "1",
    "creationDate": "2021-02-05T16:03:14.098000-08:00",
```

```
    "lastModifiedDate": "2021-02-05T16:03:14.098000-08:00",
    "generationId":
 "f19144b798534f52c619d44f771a354f1b957dfa2b850625d9f1d0fde530e75f"
}
```

6. オンラインコンバータまたはその他のツールを使用して、ポリシードキュメント文字列を JSON オブジェクトに変換し、iot-policy.json という名前のファイルに保存します。

   例えば、jq ツールがインストールされている場合には、次のコマンドを実行してポリシードキュメントを取得し、JSON オブジェクトに変換してから、ポリシードキュメントを JSON オブジェクトとして保存することができます。

   ```
   aws iot get-policy --policy-name GreengrassV2IoTThingPolicy --query
    'policyDocument' | jq fromjson >> iot-policy.json
   ```

7. 必要なアクセス許可についてポリシーを確認し、不足していれば必要なアクセス許可を追加します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを開きます。

   ```
   nano iot-policy.json
   ```

   - greengrass:PutCertificateAuthorities
   - greengrass:VerifyClientDeviceIdentity
   - greengrass:VerifyClientDeviceIoTCertificateAssociation
   - greengrass:GetConnectivityInfo
   - greengrass:UpdateConnectivityInfo - (オプション) このアクセス許可は、コアデバイスのネットワーク接続に関する情報を AWS IoT Greengrass クラウドサービスに報告する IP 検出コンポーネントを使用するために必要です。
   - iot:GetThingShadow、iot:UpdateThingShadow、および iot:DeleteThingShadow – (オプション) これらのアクセス許可は、クライアントデバイスシャドウと AWS IoT Core を同期する、シャドウマネージャーコンポーネントを使用するために必要です。この機能を使用するには、Greengrass nucleus v2.6.0 以降、シャドウマネージャー v2.2.0 以降、および MQTT ブリッジ v2.2.0 以降が必要です。

8. 変更をポリシーの新しいバージョンとして保存します。次のコマンドを実行し、*GreengrassV2IoTThingPolicy* をポリシーの名前に置き換えます。

```
aws iot create-policy-version --policy-name GreengrassV2IoTThingPolicy --policy-
document file://iot-policy.json --set-as-default
```

成功すると、次の例に類似したレスポンスが返されます。

```
{
    "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy",
    "policyDocument": "{\
  \\"Version\\": \\"2012-10-17\\",\
  \\"Statement\\": [\
    {\
      \\"Effect\\": \\"Allow\\",\
      \\"Action\\": [\
\\t\\t\\"iot:Connect\\",\
\\t\\t\\"iot:Publish\\",\
\\t\\t\\"iot:Subscribe\\",\
\\t\\t\\"iot:Receive\\",\
\\t\\t\\"greengrass:*\\"\
      ],\
      \\"Resource\\": \\"*\\"\
    }\
  ]\
}",
    "policyVersionId": "2",
    "isDefaultVersion": true
}
```

クライアントデバイスサポート用の Greengrass コンポーネント

> **Important**
>
> コアデバイスでクライアントデバイスをサポートするため、Greengrass nucleus v2.2.0 以降が実行されている必要があります

クライアントデバイスがコアデバイスに接続して通信できるようにするには、コアデバイスに次の Greengrass コンポーネントをデプロイします。

- クライアントデバイス認証 (aws.greengrass.clientdevices.Auth)

  クライアントデバイス認証コンポーネントをデプロイして、クライアントデバイスを認証し、クライアントデバイスのアクションを承認します。このコンポーネントは、AWS IoT モノがコアデバイスに接続できるようにします。

  このコンポーネントを使用するには、いくつかの設定が必要です。クライアントデバイスのグループを指定して、MQTT を介した接続や通信などの各グループが実行することができる操作を指定する必要があります。詳細については、「クライアントデバイス認証コンポーネントの設定」を参照してください。

- MQTT 3.1.1 ブローカー (モケット) (aws.greengrass.clientdevices.mqtt.Moquette)

  軽量な MQTT ブローカーを実行するため、Moquette MQTT ブローカーコンポーネントをデプロイします。Moquette MQTT ブローカーは MQTT 3.1.1 に準拠しており、QoS 0、QoS 1、QoS 2、保持されたメッセージ、Last Will メッセージ、および永続サブスクリプションに対するローカルサポートが含まれます。

  使用するにあたり、このコンポーネントを設定する必要はありません。ただし、このコンポーネントが MQTT ブローカを操作するポートを設定することができます。デフォルトではポート 8883 が使用されます。

- MQTT 5 ブローカー (EMQX) (aws.greengrass.clientdevices.mqtt.EMQX)

  > **Note**
  >
  > EMQX MQTT 5 ブローカーを使用するには、Greengrass nucleus v2.6.0 以降と、クライアントデバイスの v2.2.0 以降による認証を使用する必要があります。

  クライアントデバイスとコアデバイス間の通信で MQTT 5.0 機能を使用するために、EMQX MQTT ブローカーコンポーネントをデプロイします。EMQX MQTT ブローカーは MQTT 5.0 に準拠しており、セッションとメッセージの有効期限、ユーザープロパティ、共有サブスクリプション、トピックエイリアスなどのサポートが含まれます。

  使用するにあたり、このコンポーネントを設定する必要はありません。ただし、このコンポーネントが MQTT ブローカを操作するポートを設定することができます。デフォルトではポート 8883 が使用されます。

- MQTT ブリッジ (aws.greengrass.clientdevices.mqtt.Bridge)

(オプション) MQTT ブリッジコンポーネントをデプロイして、クライアントデバイス (ローカル MQTT)、ローカルパブリッシュ/サブスクライブ、および AWS IoT Core MQTT 間でメッセージをリレーします。クライアントデバイスを AWS IoT Core と同期するようにこのコンポーネントを設定し、Greengrass コンポーネントからクライアントデバイスとやり取りします。

このコンポーネントを使用するには、設定する必要があります。このコンポーネントがメッセージをリレーするトピックマッピングを指定する必要があります。詳細については、「MQTT ブリッジコンポーネントの設定」を参照してください。

- IP ディテクター (`aws.greengrass.clientdevices.IPDetector`)

  (オプション) IP 検出コンポーネントをデプロイして、コアデバイスの MQTT ブローカエンドポイントを AWS IoT Greengrass クラウドサービスに自動的に報告します。ルータがコアデバイスに MQTT ブローカポートを転送する場合など、複雑なネットワーク設定がある場合には、このコンポーネントを使用することはできません。

  使用するにあたり、このコンポーネントを設定する必要はありません。

- シャドウマネージャー (`aws.greengrass.ShadowManager`)

  > **Note**
  >
  > クライアントデバイスのシャドウを管理するには、Greengrass nucleus v2.6.0 以降、シャドウマネージャー v2.2.0 以降、および MQTT ブリッジ v2.2.0 以降を使用する必要があります。

  (オプション) シャドウマネージャーコンポーネントをデプロイして、コアデバイスのクライアントのデバイスシャドウを管理します。Greengrass コンポーネントは、クライアントデバイスとのやり取りのため、クライアントのデバイスシャドウを取得、更新、削除します。クライアントのデバイスシャドウを AWS IoT Core のクラウドサービスを同期するように、シャドウマネージャーコンポーネントを設定することも可能です。

  このコンポーネントをクライアントのデバイスシャドウとともに使用するには、クライアントデバイスとシャドウマネージャー間でメッセージを伝達するように、(ローカルの公開/サブスクライブを使用する) MQTT ブリッジコンポーネントを設定する必要があります。それ以外の場合、このコンポーネントの使用には設定を必要としませんが、デバイスシャドウを同期する際には設定が必要となります。

> **Note**
>
> デプロイする MQTT ブローカーコンポーネントは、1 つだけにすることをお勧めします。MQTT ブリッジと IP ディテクターコンポーネントは、一度に 1 つの MQTT ブローカーコンポーネントとのみ動作します。デプロイする MQTT ブローカーコンポーネントが複数ある場合は、それぞれが異なるポートを使用する設定を行う必要があります。

クラウドディスカバリを設定する (コンソール)

AWS IoT Greengrass コンソールを使用して、クライアントデバイスを関連付けて、コアデバイスのエンドポイントを管理し、コンポーネントをデプロイして、クライアントデバイスへのサポートを有効にすることができます。詳細については、「ステップ 2: クライアントデバイスのサポートを有効にする」を参照してください。

クラウドディスカバリを設定する (AWS CLI)

AWS Command Line Interface (AWS CLI) を使用して、クライアントデバイスを関連付けて、コアデバイスのエンドポイントを管理し、コンポーネントをデプロイして、クライアントデバイスへのサポートを有効にすることができます。詳細については、次を参照してください。

- クライアントデバイスとの関連付けを管理する (AWS CLI)
- コアデバイスのエンドポイントを管理
- AWS から提供されるクライアントデバイスコンポーネント
- デプロイの作成

クライアントデバイスを関連付ける

クラウドディスカバリを使用するには、クライアントデバイスをコアデバイスに関連付けて、コアデバイスを検出できるようにします。その後、Greengrass ディスカバリー API を使用して、関連するコアデバイスの接続情報と証明書を取得することができます。

同様に、コアデバイスを検出しないようにする場合は、クライアントデバイスのコアデバイスとの関連付けを外します。

トピック

- クライアントデバイスとの関連付けを管理する (コンソール)

- クライアントデバイスとの関連付けを管理する (AWS CLI)
- クライアントデバイスとの関連付けを管理する (API)

クライアントデバイスとの関連付けを管理する (コンソール)

AWS IoT Greengrass コンソールを使用して、クライアントデバイスとの関連付けを表示、追加、および削除することができます。

クライアントデバイスのコアデバイスとの関連付けを表示するには (コンソール)

1. AWS IoT Greengrass コンソールに移動します。
2. [Core devices] (コアデバイス) を選択します。
3. 管理するコアデバイスを選択します。
4. コアデバイスの詳細ページで、[Client devices] (クライアントデバイス) タブを選択します。
5. [Associated client devices] (関連付けられているクライアントデバイス) セクションで、どのクライアントデバイス (AWS IoT モノ) がコアデバイスに関連付けられているのかを表示できます。

クライアントデバイスをコアデバイスと関連付けるには (コンソール)

1. AWS IoT Greengrass コンソールに移動します。
2. [Core devices] (コアデバイス) を選択します。
3. 管理するコアデバイスを選択します。
4. コアデバイスの詳細ページで、[Client devices] (クライアントデバイス) タブを選択します。
5. [Associated client devices] (関連付けられているクライアントデバイス) セクションで、[Associate client devices] (クライアントデバイスを関連付ける) を選択します。
6. [Associate client devices with core device] (クライアントデバイスをコアデバイスに関連付ける) モーダルで、関連付ける各クライアントデバイスに対して次の操作を行います。

   a. クライアントデバイスとして関連付ける AWS IoT モノの名前を入力します。

   b. [追加] を選択します。
7. [関連付ける] を選択します。

   関連付けたクライアントデバイスで Greengrass ディスカバリ API を使用して、このコアデバイスを検出できるようになりました。

クライアントデバイスのコアデバイスとの関連付けを外すには (コンソール)

1. AWS IoT Greengrass コンソールに移動します。
2. [Core devices] (コアデバイス) を選択します。
3. 管理するコアデバイスを選択します。
4. コアデバイスの詳細ページで、[Client devices] (クライアントデバイス) タブを選択します。
5. [Associated client devices] (関連付けられているクライアントデバイス) セクションで、関連付けを外す各クライアントデバイスを選択します。
6. [Disassociate] (関連付け解除) を選択します。
7. 確認モーダルで、[Disassociate] (関連付け解除) を選択します。

   関連付けを外したクライアントデバイスからは、Greengrass ディスカバリ API を使用して、このコアデバイスを検出できなくなります。

クライアントデバイスとの関連付けを管理する (AWS CLI)

AWS Command Line Interface (AWS CLI) を使用して、クライアントデバイスのコアデバイスとの関連付けを管理します。

クライアントデバイスのコアデバイスとの関連付けを表示するには (AWS CLI)

- 次のコマンドを使用します: list-client-devices-associated-with-core-device。

クライアントデバイスをコアデバイスと関連付けるには (AWS CLI)

- 次のコマンドを使用します: batch-associate-client-device-with-core-device。

クライアントデバイスのコアデバイスとの関連付けを外すには (AWS CLI)

- 次のコマンドを使用します: batch-disassociate-client-device-from-core-device。

クライアントデバイスとの関連付けを管理する (API)

AWS API を使用して、クライアントデバイスのコアデバイスとの関連付けを管理することができます。

クライアントデバイスのコアデバイスとの関連付けを表示するには (AWS API)

- 次のオペレーションを使用します: ListClientDevicesAssociatedWithCoreDevice。

クライアントデバイスをコアデバイスと関連付けるには (AWS API)

- 次のオペレーションを使用します: BatchAssociateClientDeviceWithCoreDevice。

クライアントデバイスのコアデバイスとの関連付けを外すには (AWS API)

- 次のオペレーションを使用します: BatchDisassociateClientDeviceFromCoreDevice。

オフライン時のクライアントの認証

オフライン認証を使用することで、コアデバイスがクラウドに接続されていない場合でも、クライアントデバイスがコアデバイスに接続できるように、AWS IoT Greengrass Core デバイスを設定できます。オフライン認証を使用する場合、Greengrass デバイスは、部分的にオフラインの環境でも引き続き動作します。

クラウドに接続しているクライアントデバイスのためにオフライン認証を使用するには、次が必要です。

- クライアントデバイス認証 コンポーネントがデプロイされた AWS IoT Greengrass Core デバイス。オフライン認証には、バージョン 2.3.0 以降を使用する必要があります。
- クライアントデバイスの初回接続中におけるコアデバイス用のクラウド接続。

クライアントの認証情報の保存

クライアントデバイスが初めてコアデバイスに接続すると、そのコアデバイスは AWS IoT Greengrass のサービスを呼び出します。呼び出されると、Greengrass は、そのクライアントデバイスの登録を AWS IoT モノとして検証します。また、デバイスに有効な証明書があることも検証します。その後、そのコアデバイスはこの情報をローカルに保存します。

デバイスが次回接続するときに、Greengrass コアデバイスは、AWS IoT Greengrass サービスを使用してクライアントデバイスの検証を試みます。AWS IoT Greengrass に接続できない場合、コアデバイスはローカルに保存されているデバイス情報を使用して、クライアントデバイスを検証します。

Greengrass コアデバイスが認証情報を保存する期間を設定できます。タイムアウトは 1 分から 2,147,483,647 分までの間で設定できます。デフォルトは 1 分です。これにより、オフライン認証は事実上オフになります。このタイムアウトを設定するときは、セキュリティのニーズを考慮することをお勧めします。また、クラウドから切断されている間におけるコアデバイスの想定実行時間も考慮する必要があります。

コアデバイスは認証情報ストレージを 3 回更新します。

1. デバイスが初めてコアデバイスに接続するとき。
2. クライアントデバイスがコアデバイスに再接続するとき (コアデバイスがクラウドに接続されている場合)。
3. 認証情報ストア全体を更新するために 1 日に 1 回 (コアデバイスがクラウドに接続されている場合)。

Greengrass コアデバイスは、認証情報ストアを更新するときに、ListClientDevicesAssociatedWithCoreDevice オペレーションを使用します。Greengrass は、このオペレーションによって返されたデバイスのみを更新します。クライアントデバイスをコアデバイスと関連付けるには、「クライアントデバイスを関連付ける」を参照してください。

`ListClientDevicesAssociatedWithCoreDevice` オペレーションを使用するには、AWS IoT Greengrass を実行する AWS アカウント に関連付けられた AWS Identity and Access Management (IAM) ロールにオペレーションの許可を追加する必要があります。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

コアデバイスのエンドポイントを管理

クラウドディスカバリを使用するとき、コアデバイスの MQTT ブローカエンドポイントを AWS IoT Greengrass クラウドサービスに保存します。クライアントデバイスは AWS IoT Greengrass に接続して、これらのエンドポイントと関連するコアデバイスに関する他の情報を取得します。

各コアデバイスに、エンドポイントを自動または手動で管理できます。

- IP ディテクタでエンドポイントを自動的に管理

  クライアントデバイスがコアデバイスと同じネットワーク上の場所など、複雑ではないネットワーク設定を行っている場合、IP 検知コンポーネントをデプロイして、ユーザーに代わってコアデバイスエンドポイントを自動的に管理できます。例えば、コアデバイスが、MQTT ブローカーポートをコアデバイスに転送するルータの背後にある場合、IP ディテクタコンポーネントを使用できません。

IP ディテクタコンポーネントは、モノグループにあるすべてのコアデバイスのエンドポイントを管理するため、モノグループに展開する場合にも役立ちます。詳細については、「IP ディテクタを使用してエンドポイントを自動的に管理」を参照してください。

- エンドポイントを手動で管理

  IP ディテクタコンポーネントを使用できない場合、コアデバイスエンドポイントを手動で管理する必要があります。コンソールまたは API でこれらのエンドポイントを更新できます。詳細については、「エンドポイントを手動で管理」を参照してください。

トピック

IP ディテクタを使用してエンドポイントを自動的に管理

コアデバイスと同じネットワーク上でクライアントデバイスなど、単純なネットワーク設定がある場合、IP ディテクターコンポーネントをデプロイして次の手順を実行します。

- Greengrass コアデバイスのローカルネットワーク接続情報をモニタリングします。この情報には、コアデバイスのネットワークエンドポイントと MQTT ブローカーが動作するポートが含まれます。
- コアデバイスの接続情報を AWS IoT Greengrass クラウドサービスに報告します。

IP ディテクタコンポーネントは、手動で設定するエンドポイントを上書きします。

> **Important**
>
> コアデバイスの AWS IoT ポリシーは、IP ディテクタコンポーネントを使用する `greengrass:UpdateConnectivityInfo` 許可を付与する必要があります。詳細については、「データプレーンオペレーションの AWS IoT ポリシー と AWS IoTモノポリシーを設定する」を参照してください。

IP ディテクターコンポーネントをデプロイするため、次のいずれかを行います。

- コンソールの [Configure discovery] (ディスカバリの設定) ページを使用します。詳細については、「クラウドディスカバリを設定する (コンソール)」を参照してください。
- IP ディテクタを含めるようにデプロイを作成と改訂します。コンソール、AWS CLI、AWS API を使用してデプロイ管理できます。詳細については、「デプロイの作成」を参照してください。

IP ディテクタコンポーネントをデプロイ (コンソール)

1. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Components] (コンポーネント) を選択します。
2. [Components] (コンポーネント) ページで、[Public components] (公開コンポーネント) タブを選択し、次に aws.greengrass.clientdevices.IPDetector を選択します。
3. aws.greengrass.clientdevices.IPDetector ページで、[Deploy] (デプロイ) を選択します。
4. [Add to deployment] (デプロイに追加) で、改訂する既存のデプロイを選択するか、新しいデプロイを作成することを選択して、[Next] (次へ) を選択します。
5. 新しいデプロイの作成を選択した場合、デプロイのターゲットコアデバイスまたはモノグループを選択します。リポジトリの [Specify target] (ターゲットを指定) ページの、[Deployment target] (ターゲットのデプロイ) で、コアデバイスまたはモノグループを選択し、[Next] (次へ) を選択します。
6. [Select components] (コンポーネントを選択) ページで、aws.greengrass.clientdevices.IPDetector コンポーネントが選択されていることを確認し、[Next] (次) を選択します。
7. [Configure components] (コンポーネントを設定) ページで、aws.greengrass.clientdevices.IPDetector を選択したら、次の操作を行います。

   a. [Configure component] (コンポーネントを設定) を選択します。

   b. 設定aws.greengrass.clientdevices.IPDetectorモーダルの「設定更新」の「 をマージするための設定」で、設定更新を入力して IP ディテクターコンポーネントを設定できます。次の設定オプションのいずれかを指定できます。

      - `defaultPort` - (オプション) このコンポーネントが IP アドレスを検出するときに報告する MQTT ブローカーポート。デフォルトポート 8883 とは異なるポートを使用するように MQTT ブローカーを設定する場合、このパラメータを指定する必要があります。
      - `includeIPv4LoopbackAddrs` - (オプション) このオプションを有効にして IPv4 ループバックアドレスを検出して報告します。これらは、`localhost` など、IP アドレスであ

り、デバイスが自身と通信できる場所です。このオプションは、コアデバイスとクライアントデバイスを同じシステムで実行するテスト環境で使用します。

- `includeIPv4LinkLocalAddrs` - (オプション) このオプションを有効にして IPv4 リンクローカルアドレスを検出して報告します。このオプションは、コアデバイスのネットワークに Dynamic Host Configuration Protocol (DHCP) または静的に割り当てられた IP アドレスがない場合に使用します。

設定更新は、次の例のようになることがあります。

```
{
  "defaultPort": "8883",
  "includeIPv4LoopbackAddrs": false,
  "includeIPv4LinkLocalAddrs": false
}
```

c. [Confirm] (確認) を選択してモーダルを閉じ、次に [Next] (次) を選択します。

8. [Configure advanced settings] (詳細設定) ページはデフォルト設定のままにし、[Next] (次へ) を選択します。
9. [Review] ページで、[デプロイ] を選択します。

   デプロイに最大 1 分かかる場合があります。

IP ディテクタコンポーネント (AWS CLI) の展開

IP ディテクタコンポーネントをデプロイするには、`components` オブジェクトに `aws.greengrass.clientdevices.IPDetector` を含むデプロイドキュメントを作成してコンポーネントの設定更新を指定します。デプロイの作成 の指示に従って、新しいデプロイを作成または既存のデプロイを改訂します。

デプロイドキュメントの作成時に IP ディテクタコンポーネントを設定するため、次のオプションのいずれかを指定できます。

- `defaultPort` - (オプション) このコンポーネントが IP アドレスを検出するときに報告する MQTT ブローカーポート。デフォルトポート 8883 とは異なるポートを使用するように MQTT ブローカーを設定する場合、このパラメータを指定する必要があります。
- `includeIPv4LoopbackAddrs` - (オプション) このオプションを有効にして IPv4 ループバックアドレスを検出して報告します。これらは、`localhost` など、IP アドレスであり、デバイスが自身

と通信できる場所です。このオプションは、コアデバイスとクライアントデバイスを同じシステムで実行するテスト環境で使用します。

- `includeIPv4LinkLocalAddrs` - (オプション) このオプションを有効にして IPv4 リンクローカルアドレスを検出して報告します。このオプションは、コアデバイスのネットワークに Dynamic Host Configuration Protocol (DHCP) または静的に割り当てられた IP アドレスがない場合に使用します。

次の部分的なデプロイドキュメントの例では、ポート 8883 を MQTT ブローカーポートとして報告するように指定しています。

```
{
  ...,
  "components": {
    ...,
    "aws.greengrass.clientdevices.IPDetector": {
      "componentVersion": "2.1.1",
      "configurationUpdate": {
        "merge": "{\"defaultPort\":\"8883\",}"
      }
    }
  }
}
```

エンドポイントを手動で管理

コアデバイスの MQTT ブローカーエンドポイントを手動で管理できます。

各 MQTT ブローカーのエンドポイントには次の情報があります。

[Endpoint] (エンドポイント) (`HostAddress`)

クライアントデバイスがコアデバイスの MQTT ブローカーに接続できる IP アドレスまたは DNS アドレス。

ポート (`PortNumber`)

MQTT ブローカーがコアデバイスで動作するポート。

このポートは、モケット MQTT ブローカーコンポーネントに設定でき、デフォルトでポート 8883 が使用されます。

[Metadata] (メタデータ) (`Metadata`)

このエンドポイントに接続するクライアントデバイスに提供する追加のメタデータ。

トピック

- エンドポイントの管理 (コンソール)
- エンドポイント (AWS CLI) を管理します。
- エンドポイント (API) の管理

エンドポイントの管理 (コンソール)

AWS IoT Greengrass コンソールを使用して、コアデバイスのエンドポイントを確認、更新、削除します。

コアデバイス (コンソール) のエンドポイントを管理するには

1. AWS IoT Greengrass コンソールに移動します。
2. [Core devices] (コアデバイス) を選択します。
3. 管理するコアデバイスを選択します。
4. コアデバイスの詳細ページで、[Client devices] (クライアントデバイス) タブを選択します。
5. [MQTT broker endpoints] (MQTT ブローカーエンドポイント) セクションで、コアデバイスの MQTT ブローカーエンドポイントを確認できます。[Manage endpoints] (エンドポイント管理) を選択します。
6. [Manage endpoints] (エンドポイント管理) モーダルで、コアデバイスの MQTT ブローカーエンドポイントを追加または削除します。
7. [更新] を選択します。

エンドポイント (AWS CLI) を管理します。

AWS Command Line Interface (AWS CLI) を使用してコアデバイスのエンドポイントを管理します。

> **Note**
>
> AWS IoT Greengrass V2 に対するクライアントデバイスのサポートは、AWS IoT Greengrass V1 と後方互換性があるため、AWS IoT Greengrass V2 または AWS IoT Greengrass V1 API 操作を使用してコアデバイスのエンドポイントを管理できます。

コアデバイス (AWS CLI) のエンドポイントを取得するには

- 次のコマンドのいずれかを実行してください:
  - greengrassv2: get-connectivity-info
  - greengrass: get-connectivity-info

コアデバイス (AWS CLI) のエンドポイントを更新するには

- 次のコマンドのいずれかを実行してください:
  - greengrassv2: update-connectivity-info
  - greengrass: update-connectivity-info

エンドポイント (API) の管理

AWS API を使用してコアデバイスのエンドポイントを管理できます。

> **Note**
>
> AWS IoT Greengrass V2 に対するクライアントデバイスのサポートは、AWS IoT Greengrass V1 と後方互換性があるため、AWS IoT Greengrass V2 または AWS IoT Greengrass V1 API 操作を使用してコアデバイスのエンドポイントを管理できます。

コアデバイス (AWS API) のエンドポイントを取得するには

- 次の操作のいずれかを実行してください:
  - V2: GetConnectivityInfo
  - V1: GetConnectivityInfo

コアデバイス (AWS API) のエンドポイントを更新するには

- 次の操作のいずれかを実行してください:
  - V2: UpdateConnectivityInfo
  - V1: UpdateConnectivityInfo

MQTT ブローカーを選択する

AWS IoT Greengrass では、コアデバイスで実行するローカル MQTT ブローカーを、選択するためのオプションを用意しています。クライアントデバイスは、コアデバイスで実行される MQTT ブローカーに接続します。したがって、そのクライアントデバイスと互換性のある MQTT ブローカーを選択する必要があります。

> **Note**
>
> デプロイする MQTT ブローカーコンポーネントは、1 つだけにすることをお勧めします。MQTT ブリッジと IP ディテクターコンポーネントは、一度に 1 つの MQTT ブローカーコンポーネントとのみ動作します。デプロイする MQTT ブローカーコンポーネントが複数ある場合は、それぞれが異なるポートを使用する設定を行う必要があります。

MQTT ブローカーは、以下から選択できます。

- MQTT 3.1.1 ブローカー (Moquette) – `aws.greengrass.clientdevices.mqtt.Moquette`

  MQTT 3.1.1 標準に準拠した軽量 MQTT ブローカーの場合は、このオプションを選択します。AWS IoT Core MQTT ブローカーと AWS IoT Device SDK も MQTT 3.1.1 標準にも準拠しているため、これらの機能を使用して、デバイス全体と AWS クラウド で MQTT 3.1.1 を使用するアプリケーションを作成ことが可能です。
- MQTT 5 ブローカー (EMQX) – `aws.greengrass.clientdevices.mqtt.EMQX`

  コアデバイスとクライアントデバイス間の通信に MQTT 5 の機能を使用するには、このオプションを選択します。このコンポーネントは、Mocquette MQTT 3.1.1 ブローカーより多くのリソースを使用します。また、Linuxコアデバイスでは Docker が必要となります。

  MQTT 5 は MQTT 3.1.1 に対し下位互換性があるため、このブローカーにも、MQTT 3.1.1 を使用するクライアントデバイスを接続することができます。実行している Mocquette MQTT 3.1.1 ブローカーを EMQX MQTT 5ブローカーに置き換えることが可能で、その場合も、クライアントデバイスは問題なく接続や動作を継続できます。
- カスタムブローカーを実装する

  クライアントデバイスと通信する、カスタムのローカルブローカーコンポーネントを作成するには、このオプションを選択します。カスタムのローカルブローカーを作成すると、MQTT 以外のプロトコルを使用できます。AWS IoT Greengrass には、クライアントデバイスの認証と承認に使

用できる、コンポーネント SDK が用意されています。詳細については、「AWS IoT Device SDK を使用して Greengrass nucleus、その他のコンポーネント、および AWS IoT Core と通信する」および「クライアントデバイスを認証して承認する」を参照してください。

MQTT ブローカーを使用したクライアントデバイスの AWS IoT Greengrass Core デバイスへの接続

AWS IoT Greengrass Core デバイスで MQTT ブローカーを使用する場合、そのデバイスは、自らに固有のコアデバイス認証機関 (CA) を使用して、クライアントとの相互 TLS 接続を確立するためにブローカーに証明書を発行します。

AWS IoT Greengrass はコアデバイス CA を自動生成しますが、独自のものを提供することもできます。クライアントデバイス認証 コンポーネントが接続されると、コアデバイス CA が AWS IoT Greengrass に登録されます。自動生成されたコアデバイス CA は永続的であり、クライアントデバイス認証コンポーネントが設定されている限り、デバイスは同じ CA を使用し続けます。

MQTT ブローカーが起動すると、証明書をリクエストします。クライアントデバイス認証コンポーネントは、コアデバイス CA を使用して X.509 証明書を発行します。証明書は、ブローカーの起動時、証明書の失効時、または IP アドレスなどの接続に関する情報の変更時にローテーションされます。詳細については、「ローカル MQTT ブローカー上での証明書ローテーション」を参照してください。

クライアントを MQTT ブローカーに接続するには、次が必要です。

- クライアントデバイスには AWS IoT Greengrass Core デバイス CA が必要です。この CA は、クラウド検出を通じて、または手動で CA を提供することによって取得できます。詳細については、「独自の認証機関の使用」を参照してください。
- コアデバイスの完全修飾ドメイン名 (FQDN) または IP アドレスは、コアデバイス CA によって発行されたブローカー証明書に存在する必要があります。IP ディテクター コンポーネントを使用するか、または IP アドレスを手動で設定することで、これを確実に実現できます。詳細については、「コアデバイスのエンドポイントを管理」を参照してください。
- クライアントデバイス認証コンポーネントは、Greengrass コアデバイスに接続するための許可をクライアントデバイスに付与する必要があります。詳細については、「クライアントデバイス認証」を参照してください。

独自の認証機関の使用

クライアントデバイスがクラウドにアクセスしてコアデバイスを検出できない場合は、コアデバイス認証機関 (CA) を指定できます。Greengrass コアデバイスは、コアデバイス CA を使用して MQTT ブローカーの証明書を発行します。コアデバイスを設定してクライアントデバイスに CA をプロビジョニングすると、クライアントデバイスは、コアデバイス CA (独自に提供された CA または自動生成された CA) を使用してエンドポイントに接続し、TLS ハンドシェイクを検証できます。

コアデバイス CA を使用するように クライアントデバイス認証 コンポーネントを設定するには、コンポーネントをデプロイするときに `certificateAuthority` 設定パラメータを設定します。設定時には次の詳細情報を入力する必要があります。

- コアデバイス CA 証明書の場所。
- コアデバイス CA 証明書のプライベートキー。
- (オプション) ルート証明書への証明書チェーン (コアデバイス CA が中間 CA の場合)。

コアデバイス CA を提供する場合、AWS IoT Greengrass はその CA をクラウドに登録します。

証明書は、ハードウェアセキュリティモジュールまたはファイルシステムに保存できます。次の例は、HSM/TPM を使用して格納された中間 CA の `certificateAuthority` 設定を示しています。証明書チェーンはディスクにのみ保存できることに注意してください。

```
"certificateAuthority": {
    "certificateUri": "pkcs11:object=CustomerIntermediateCA;type=cert",
    "privateKeyUri": "pkcs11:object=CustomerIntermediateCA;type=private"
    "certificateChainUri": "file:///home/ec2-user/creds/certificateChain.pem",
  }
```

この例では、`certificateAuthority` 設定パラメータがファイルシステムから中間 CA を使用するように、クライアントデバイス認証コンポーネントを設定します。

```
"certificateAuthority": {
    "certificateUri": "file:///home/ec2-user/creds/intermediateCA.pem",
    "privateKeyUri": "file:///home/ec2-user/creds/intermediateCA.privateKey.pem",
    "certificateChainUri": "file:///home/ec2-user/creds/certificateChain.pem",
  }
```

デバイスを AWS IoT Greengrass Core デバイスに接続するには、次を実行します。

1. 組織のルート CA を使用して Greengrass コアデバイス用の中間認証機関 (CA) を作成します。セキュリティに関するベストプラクティスとして、中間 CA を使用することをお勧めします。
2. 中間 CA 証明書、プライベートキー、証明書チェーンを、ルート CA から Greengrass コアデバイスに提供します。詳細については、「クライアントデバイス認証」を参照してください。中間 CA は Greengrass コアデバイスのコアデバイス CA となり、デバイスはその CA を AWS IoT Greengrass に登録します。
3. クライアントデバイスを AWS IoT モノとして登録します。詳細については、「AWS IoT Core デベロッパーガイド」の「モノのオブジェクトを作成する」を参照してください。プライベートキー、パブリックキー、デバイス証明書、およびルート CA 証明書をクライアントデバイスに追加します。情報を追加する方法は、デバイスとソフトウェアによって異なります。

デバイスを設定すると、証明書とパブリックキーチェーンを使用して Greengrass コアデバイスに接続できるようになります。コアデバイスエンドポイントを見つけるのはソフトウェアの役割です。コアデバイスのエンドポイントを手動で設定できます。詳細については、「エンドポイントを手動で管理」を参照してください。

クライアントデバイス通信をテストする

クライアントデバイスでは AWS IoT Device SDK を使用して、コアデバイスを検出して接続し、通信することができます。AWS IoT Device SDK の Greengrass 検出クライアントを使用することで、クライアントデバイスが接続可能なコアデバイスに関する情報を返す [Greengrass discovery API] (Greengrass 検出 API) を使用できます。API レスポンスには、接続する MQTT ブローカエンドポイントと、各コアデバイスの身元を確認するために使用する証明書が含まれます。その後、クライアントデバイスは、コアデバイスに正常に接続されるまで、各エンドポイントを試すことができます。

クライアントデバイスは、関連付けられているコアデバイスのみを検出できます。クライアントデバイスとコアデバイス間の通信をテストする前に、クライアントデバイスをコアデバイスに関連付ける必要があります。詳細については、「クライアントデバイスを関連付ける」を参照してください。

Greengrass 検出 API は、指定したコアデバイス MQTT ブローカーエンドポイントを返します。[IP detector component] (IP ディテクターコンポーネント) を使用して、これらのエンドポイントを管理することも、各コアデバイスに対して手動で管理することもできます。詳細については、「コアデバイスのエンドポイントを管理」を参照してください。

> **Note**
>
> Greengrass discovery API を使用するには、クライアントデバイスに `greengrass:Discover` アクセス許可が必要です。詳細については、「クライアントデバイス向けの最低限の AWS IoT ポリシー」を参照してください。

AWS IoT Device SDK は、複数のプログラミング言語で利用できます。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT デバイス SDK」を参照してください。

トピック

- 通信をテストする (Python)
- 通信をテストする (C++)
- 通信をテストする (JavaScript)
- 通信をテストする (Java)

通信をテストする (Python)

このセクションでは、AWS IoT Device SDK v2 for Python の Greengrass 検出サンプルを使用して、クライアントデバイスとコアデバイス間の通信をテストします。

> **Important**
>
> AWS IoT Device SDK v2 for Python を使用するには、デバイスは Python 3.6 以降を実行する必要があります。

通信をテストするには (AWS IoT Device SDK v2 for Python)

1. AWS IoT Device SDK v2 for Python をクライアントデバイスとして接続する AWS IoT モノにダウンロードしてインストールします。

   クライアントデバイスで、次の操作を行います。

   a. AWS IoT Device SDK v2 for Python のリポジトリをクローンしてダウンロードします。

   ```
   git clone https://github.com/aws/aws-iot-device-sdk-python-v2.git
   ```

b. AWS IoT Device SDK v2 for Python をインストールします。

```
python3 -m pip install --user ./aws-iot-device-sdk-python-v2
```

2. AWS IoT Device SDK v2 for Python のサンプルフォルダに移動します。

```
cd aws-iot-device-sdk-python-v2/samples
```

3. サンプルの Greengrass 検出アプリケーションを実行します。このアプリケーションでは、クライアントデバイスのモノの名前、使用する MQTT トピックとメッセージ、および接続を認証して保護する証明書を指定する引数が必要です。次の例では、clients/*MyClientDevice1*/hello/world トピックに「Hello World」メッセージを送信します。

   - *MyClientDevice1* をクライアントデバイスのモノの名前に置き換えます。
   - *~/certs/AmazonRootCA1.pem* をクライアントデバイスの Amazon ルート CA 証明書へのパスに置き換えます。
   - *~/certs/device.pem.crt* をクライアントデバイスのデバイス証明書へのパスに置き換えます。
   - *~/certs/private.pem.key* をクライアントデバイスのプライベートキーファイルへのパスに置き換えます。
   - *us-east-1* をクライアントデバイスとコアデバイスが動作する AWS リージョンに置き換えます。

```
python3 basic_discovery.py \\
  --thing_name MyClientDevice1 \\
  --topic 'clients/MyClientDevice1/hello/world' \\
  --message 'Hello World!' \\
  --ca_file ~/certs/AmazonRootCA1.pem \\
  --cert ~/certs/device.pem.crt \\
  --key ~/certs/private.pem.key \\
  --region us-east-1 \\
  --verbosity Warn
```

検出サンプルアプリケーションはメッセージを 10 回送信し、その後切断します。また、メッセージの公開先と同じトピックにサブスクライブします。出力にアプリケーションがトピックに関する MQTT メッセージを受信したことが示される場合は、クライアントデバイスはコアデバイスと正常に通信できています。

```
Performing greengrass discovery...
awsiot.greengrass_discovery.DiscoverResponse(gg_groups=[awsiot.greengrass_discovery.GGGroup
coreDevice-MyGreengrassCore',
 cores=[awsiot.greengrass_discovery.GGCore(thing_arn='arn:aws:iot:us-
east-1:123456789012:thing/MyGreengrassCore',
 connectivity=[awsiot.greengrass_discovery.ConnectivityInfo(id='203.0.113.0',
 host_address='203.0.113.0', metadata='', port=8883)])],
 certificate_authorities=['-----BEGIN CERTIFICATE-----\
MIICiT...EXAMPLE=\
-----END CERTIFICATE-----\
'])])
Trying core arn:aws:iot:us-east-1:123456789012:thing/MyGreengrassCore at host
 203.0.113.0 port 8883
Connected!
Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!",
 "sequence": 0}

Publish received on topic clients/MyClientDevice1/hello/world
b'{"message": "Hello World!", "sequence": 0}'
Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!",
 "sequence": 1}

Publish received on topic clients/MyClientDevice1/hello/world
b'{"message": "Hello World!", "sequence": 1}'

...

Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!",
 "sequence": 9}

Publish received on topic clients/MyClientDevice1/hello/world
b'{"message": "Hello World!", "sequence": 9}'
```

アプリケーションが代わりにエラーを出力する場合は、「Greengrass 検出で生じる問題のトラブルシューティング」を参照してください。

コアデバイスの Greengrass ログを表示して、クライアントデバイスが正常に接続してメッセージを送信しているかどうかを確認することもできます。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

通信をテストする (C++)

このセクションでは、AWS IoT Device SDK v2 for C++ の Greengrass 検出サンプルを使用して、クライアントデバイスとコアデバイス間の通信をテストします。

AWS IoT Device SDK v2 for C++ をビルドするには、デバイスに以下のツールが必要です。

- C++ 11 以降
- CMake 3.1 以降
- 以下のいずれかのコンパイラ:
  - GCC 4.8 以降
  - Clang 3.9 以降
  - MSVC 2015 以降

通信をテストするには (AWS IoT Device SDK v2 for C++)

1. AWS IoT Device SDK v2 for C++ をクライアントデバイスとして接続する AWS IoT モノにダウンロードしてビルドします。

   クライアントデバイスで、次の操作を行います。

   a. AWS IoT Device SDK v2 for C++ のワークスペース用のフォルダを作成し、そのフォルダに変更します。

   ```
   cd
   mkdir iot-device-sdk-cpp
   cd iot-device-sdk-cpp
   ```

   b. AWS IoT Device SDK v2 for C++ のリポジトリをクローンしてダウンロードします。`--recursive` フラグは、サブモジュールをダウンロードすることを指定します。

   ```
   git clone --recursive https://github.com/aws/aws-iot-device-sdk-cpp-v2.git
   ```

   c. AWS IoT Device SDK v2 for C++ のビルド出力用のフォルダを作成し、そのフォルダに変更します。

   ```
   mkdir aws-iot-device-sdk-cpp-v2-build
   cd aws-iot-device-sdk-cpp-v2-build
   ```

d. AWS IoT Device SDK v2 for C++ をビルドします。

```
cmake -DCMAKE_INSTALL_PREFIX="~/iot-device-sdk-cpp" -
DCMAKE_BUILD_TYPE="Release" ../aws-iot-device-sdk-cpp-v2
cmake --build . --target install
```

2. Greengrass 検出サンプルアプリケーションを AWS IoT Device SDK v2 for C++ でビルドします。以下の操作を実行します。

a. AWS IoT Device SDK v2 for C++ の Greengrass 検出サンプルフォルダに変更します。

```
cd ../aws-iot-device-sdk-cpp-v2/samples/greengrass/basic_discovery
```

b. Greengrass 検出サンプルビルド出力用のフォルダを作成し、そのフォルダに変更します。

```
mkdir build
cd build
```

c. Greengrass 検出サンプルアプリケーションをビルドします。

```
cmake -DCMAKE_PREFIX_PATH="~/iot-device-sdk-cpp" -
DCMAKE_BUILD_TYPE="Release" ..
cmake --build . --config "Release"
```

3. サンプルの Greengrass 検出アプリケーションを実行します。このアプリケーションでは、クライアントデバイスのモノの名前、使用する MQTT トピック、および接続を認証して保護する証明書を指定する引数が必要です。次の例では、`clients/`*`MyClientDevice1`*`/hello/world` トピックをサブスクライブし、コマンドラインで入力したメッセージを同じトピックに公開します。

- *`MyClientDevice1`* をクライアントデバイスのモノの名前に置き換えます。
- *`~/certs/AmazonRootCA1.pem`* をクライアントデバイスの Amazon ルート CA 証明書へのパスに置き換えます。
- *`~/certs/device.pem.crt`* をクライアントデバイスのデバイス証明書へのパスに置き換えます。
- *`~/certs/private.pem.key`* をクライアントデバイスのプライベートキーファイルへのパスに置き換えます。
- *`us-east-1`* をクライアントデバイスとコアデバイスが動作する AWS リージョンに置き換えます。

```
./basic-discovery \
  --thing_name MyClientDevice1 \
  --topic 'clients/MyClientDevice1/hello/world' \
  --ca_file ~/certs/AmazonRootCA1.pem \
  --cert ~/certs/device.pem.crt \
  --key ~/certs/private.pem.key \
  --region us-east-1
```

検出サンプルアプリケーションは、トピックをサブスクライブし、公開するメッセージを入力するよう促します。

```
Connecting to group greengrassV2-coreDevice-MyGreengrassCore with thing arn
 arn:aws:iot:us-east-1:123456789012:thing/MyGreengrassCore, using endpoint
 203.0.113.0:8883
Connected to group greengrassV2-coreDevice-MyGreengrassCore, using connection to
 203.0.113.0:8883
Successfully subscribed to clients/MyClientDevice1/hello/world
Enter the message you want to publish to topic clients/MyClientDevice1/hello/world
 and press enter. Enter 'exit' to exit this program.
```

アプリケーションが代わりにエラーを出力する場合は、「Greengrass 検出で生じる問題のトラブルシューティング」を参照してください。

4. **Hello World!** などのメッセージを入力します。

```
Enter the message you want to publish to topic clients/MyClientDevice1/hello/world
 and press enter. Enter 'exit' to exit this program.
Hello World!
```

出力にアプリケーションがトピックに関する MQTT メッセージを受信したことが示される場合は、クライアントデバイスはコアデバイスと正常に通信できています。

```
Operation on packetId 2 Succeeded
Publish received on topic clients/MyClientDevice1/hello/world
Message:
Hello World!
```

コアデバイスの Greengrass ログを表示して、クライアントデバイスが正常に接続してメッセージを送信しているかどうかを確認することもできます。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

通信をテストする (JavaScript)

このセクションでは、 AWS IoT Device SDKv2 JavaScriptの Greengrass 検出サンプルを使用して、クライアントデバイスとコアデバイス間の通信をテストします。

> **Important**
>
> v2 AWS IoT Device SDK for を使用するには JavaScript、デバイスは Node v10.0 以降を実行する必要があります。

通信をテストするには (AWS IoT Device SDK v2 for JavaScript)

1. AWS IoT Device SDK v2 for JavaScriptをクライアントデバイスとして接続するAWS IoTモノにダウンロードしてインストールします。

   クライアントデバイスで、次の操作を行います。

   a. v2 AWS IoT Device SDK for JavaScript リポジトリを複製してダウンロードします。

   ```
   git clone https://github.com/aws/aws-iot-device-sdk-js-v2.git
   ```

   b. AWS IoT Device SDK v2 for をインストールします JavaScript。

   ```
   cd aws-iot-device-sdk-js-v2
   npm install
   ```

2. v2 for の Greengrass AWS IoT Device SDK 検出サンプルフォルダに変更します JavaScript。

   ```
   cd samples/node/basic_discovery
   ```

3. Greengrass 検出サンプルアプリケーションをインストールします。

   ```
   npm install
   ```

4. サンプルの Greengrass 検出アプリケーションを実行します。このアプリケーションでは、クライアントデバイスのモノの名前、使用する MQTT トピックとメッセージ、および接続を認証して保護する証明書を指定する引数が必要です。次の例では、clients/*MyClientDevice1*/hello/world トピックに「Hello World」メッセージを送信します。

   - *MyClientDevice1* をクライアントデバイスのモノの名前に置き換えます。
   - *~/certs/AmazonRootCA1.pem* をクライアントデバイスの Amazon ルート CA 証明書へのパスに置き換えます。
   - *~/certs/device.pem.crt* をクライアントデバイスのデバイス証明書へのパスに置き換えます。
   - *~/certs/private.pem.key* をクライアントデバイスのプライベートキーファイルへのパスに置き換えます。
   - *us-east-1* をクライアントデバイスとコアデバイスが動作する AWS リージョンに置き換えます。

```
node dist/index.js \
  --thing_name MyClientDevice1 \
  --topic 'clients/MyClientDevice1/hello/world' \
  --message 'Hello World!' \
  --ca_file ~/certs/AmazonRootCA1.pem \
  --cert ~/certs/device.pem.crt \
  --key ~/certs/private.pem.key \
  --region us-east-1 \
  --verbose warn
```

検出サンプルアプリケーションはメッセージを 10 回送信し、その後切断します。また、メッセージの公開先と同じトピックにサブスクライブします。出力にアプリケーションがトピックに関する MQTT メッセージを受信したことが示される場合は、クライアントデバイスはコアデバイスと正常に通信できています。

```
Discovery Response:
{"gg_groups":[{"gg_group_id":"greengrassV2-coreDevice-
MyGreengrassCore","cores":[{"thing_arn":"arn:aws:iot:us-
east-1:123456789012:thing/MyGreengrassCore","connectivity":
[{"id":"203.0.113.0","host_address":"203.0.113.0","port":8883,"metadata":""}]}],"certificat
["-----BEGIN CERTIFICATE-----\nMIICiT...EXAMPLE=\n-----END CERTIFICATE-----\n"]}]}
Trying
 endpoint={"id":"203.0.113.0","host_address":"203.0.113.0","port":8883,"metadata":""}
```

```
[WARN] [2021-06-12T00:46:45Z] [00007f90c0e8d700] [socket] - id=0x7f90b8018710
 fd=26: setsockopt() for NO_SIGNAL failed with errno 92. If you are having SIGPIPE
 signals thrown, you may want to install a signal trap in your application layer.
Connected to
 endpoint={"id":"203.0.113.0","host_address":"203.0.113.0","port":8883,"metadata":""}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":1}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":2}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":3}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":4}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":5}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":6}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":7}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":8}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":9}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":10}
Complete!
```

アプリケーションが代わりにエラーを出力する場合は、「Greengrass 検出で生じる問題のトラブルシューティング」を参照してください。

コアデバイスの Greengrass ログを表示して、クライアントデバイスが正常に接続してメッセージを送信しているかどうかを確認することもできます。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

通信をテストする (Java)

このセクションでは、AWS IoT Device SDK v2 for Java の Greengrass 検出サンプルを使用して、クライアントデバイスとコアデバイス間の通信をテストします。

> **Important**
>
> AWS IoT Device SDK V2 for Java をビルドするには、デバイスには次のツールが必要です。
>
> - Java 8 以降、`JAVA_HOME` が Java フォルダを指していること。
> - Apache Maven

通信をテストするには (AWS IoT Device SDK v2 for Java)

1. AWS IoT Device SDK v2 for Java をクライアントデバイスとして接続する AWS IoT モノにダウンロードしてビルドします。

   クライアントデバイスで、次の操作を行います。

   a. AWS IoT Device SDK v2 for Java のリポジトリをクローンしてダウンロードします。

   ```
   git clone https://github.com/aws/aws-iot-device-sdk-java-v2.git
   ```

   b. AWS IoT Device SDK v2 for Javaフォルダに変更します。

   c. AWS IoT Device SDK v2 for Java をビルドします。

   ```
   cd aws-iot-device-sdk-java-v2
   mvn versions:use-latest-versions -Dincludes="software.amazon.awssdk.crt*"
   mvn clean install
   ```

2. サンプルの Greengrass 検出アプリケーションを実行します。このアプリケーションでは、クライアントデバイスのモノの名前、使用する MQTT トピック、および接続を認証して保護する証明書を指定する引数が必要です。次の例では、`clients/`*`MyClientDevice1`*`/hello/world` トピックをサブスクライブし、コマンドラインで入力したメッセージを同じトピックに公開します。

   - *`MyClientDevice1`* の両方のインスタンスをクライアントデバイスのモノの名前に置き換えます。

- *$HOME/certs/AmazonRootCA1.pem* をクライアントデバイスの Amazon ルート CA 証明書へのパスに置き換えます。
- *$HOME/certs/device.pem.crt* をクライアントデバイスのデバイス証明書へのパスに置き換えます。
- *$HOME/certs/private.pem.key* をクライアントデバイスのプライベートキーファイルへのパスに置き換えます。
- *us-east-1* をクライアントデバイスとコアデバイスが動作する AWS リージョン に置き換えます。

```
DISCOVERY_SAMPLE_ARGS="--thing_name MyClientDevice1 \
  --topic 'clients/MyClientDevice1/hello/world' \
  --ca_file $HOME/certs/AmazonRootCA1.pem \
  --cert $HOME/certs/device.pem.crt \
  --key $HOME/certs/private.pem.key \
  --region us-east-1"

mvn exec:java -pl samples/Greengrass \
  -Dexec.mainClass=greengrass.BasicDiscovery \
  -Dexec.args="$DISCOVERY_SAMPLE_ARGS"
```

検出サンプルアプリケーションは、トピックをサブスクライブし、公開するメッセージを入力するよう促します。

```
Connecting to group ID greengrassV2-coreDevice-MyGreengrassCore, with thing
 arn arn:aws:iot:us-east-1:123456789012:thing/MyGreengrassCore, using endpoint
 203.0.113.0:8883
Started a clean session
Enter the message you want to publish to topic clients/MyClientDevice1/hello/world
 and press Enter. Type 'exit' or 'quit' to exit this program:
```

アプリケーションが代わりにエラーを出力する場合は、「Greengrass 検出で生じる問題のトラブルシューティング」を参照してください。

3. **Hello World!** などのメッセージを入力します。

```
Enter the message you want to publish to topic clients/MyClientDevice1/hello/world
 and press Enter. Type 'exit' or 'quit' to exit this program:
Hello World!
```

出力にアプリケーションがトピックに関する MQTT メッセージを受信したことが示される場合は、クライアントデバイスはコアデバイスと正常に通信できています。

```
Message received on topic clients/MyClientDevice1/hello/world: Hello World!
```

コアデバイスの Greengrass ログを表示して、クライアントデバイスが正常に接続してメッセージを送信しているかどうかを確認することもできます。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

Greengrass Discovery RESTful API

AWS IoT Greengrass は、クライアントデバイスが接続できる Greengrass コアデバイスを識別するために使用できる `Discover` API オペレーションを提供します。クライアントデバイスは、このデータプレーンオペレーションを使用して、BatchAssociateClientDeviceWithCoreDeviceGreengrass コアデバイスに接続するために必要な情報を取得し、 API オペレーションに関連付けます。クライアントデバイスがオンラインになると、AWS IoT Greengrass クラウドサービスに接続し、Discovery API を使用して以下を検索できます。

- 関連付けられている各 Greengrass コアデバイスの IP アドレスとポート。
- Greengrass コアデバイスを認証するために使用できる、コアデバイスの CA 証明書。

> **Note**
>
> クライアントデバイスでも、AWS IoT Device SDK Greengrass コアデバイス検出クライアントを使用して Greengrass コアデバイスの接続情報を検出できます。検出クライアントは検出 API を使用します。詳細については、次を参照してください。
>
> - クライアントデバイス通信をテストする
> - 「AWS IoT Greengrass Version 1 デベロッパーガイド」の「Greengrass Discovery RESTful API」。

この API 操作を使用するには、Greengrass データプレーンのエンドポイントの 検出 API に HTTP リクエストを送信します。この API エンドポイントの形式は次のとおりです。

```
https://greengrass-ats.iot.region.amazonaws.com:port/greengrass/discover/thing/thing-name
```

AWS IoT Greengrass 検出 API のエンドポイントおよび AWS リージョン の一覧については、「AWS 全般のリファレンス」の「AWS IoT Greengrass V2 エンドポイントとクオータ」を参照してください。この API 操作は Greengrass データプレーンのエンドポイントでのみ利用できます。コンポーネントおよびデプロイの管理に使用するコントロールプレーンのエンドポイントは、データプレーンのエンドポイントとは異なります。

> **Note**
>
> AWS IoT Greengrass V1 と AWS IoT Greengrass V2 の検出 API は同じです。AWS IoT Greengrass V1 コアに接続するクライアントデバイスがある場合、クライアントデバイスのコードを変更することなく、それらを AWS IoT Greengrass V2 コアデバイスに接続することができます。詳細については、「AWS IoT Greengrass Version 1 デベロッパーガイド」の「Greengrass Discovery RESTful API」を参照してください。

トピック

検出認証と認可

検出 API を使用して接続情報を取得するには、クライアントデバイスで X.509 クライアント証明書による TLS 相互認証を使用して認証する必要があります。詳細については、「AWS IoT Core デベロッパーガイド」の「X.509 クライアント証明書」を参照してください。

クライアントデバイスには、`greengrass:Discover` アクションを実行する権限がある必要があります。以下の AWS IoT ポリシーの例は、`MyClientDevice1` という名前の AWS IoT モノに `Discover` の実行が許可されています。

```
{
  "Version": "2012-10-17",
```

```
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "greengrass:Discover",
      "Resource": [
        "arn:aws:iot:us-west-2:123456789012:thing/MyClientDevice1"
      ]
    }
  ]
}
```

Important

モノのポリシー変数 (`iot:Connection.Thing.*`) は コアデバイスまたは Greengrass データプレーン操作用の AWS IoT ポリシーではサポートされていません。代わりに、ワイルドカードを使用して名前が似ている複数のデバイスと一致させることができます。たとえば、`MyGreengrassDevice*` と指定すると `MyGreengrassDevice1`、`MyGreengrassDevice2` などと一致します。

詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT Core ポリシー」を参照してください。

リクエスト

次の例で示すように、リクエストにはスタンダードな HTTP ヘッダーが含まれ、Greengrass 検出エンドポイントに送信されます。

ポート番号は、コアデバイスがポート 8443 またはポート 443 のどちらで HTTPS トラフィックを送信するように設定されているかによって異なります。詳細については、「the section called “ポート 443 での接続またはネットワークプロキシを通じた接続”」を参照してください。

Note

この例では、ATS ルート CA 証明書 (推奨) で機能する Amazon Trust Services (ATS) エンドポイントを使用します。エンドポイントはルート CA 証明書タイプと一致する必要があります。

ポート 8443

```
HTTP GET https://greengrass-ats.iot.region.amazonaws.com:8443/greengrass/discover/
thing/thing-name
```

ポート 443

```
HTTP GET https://greengrass-ats.iot.region.amazonaws.com:443/greengrass/discover/
thing/thing-name
```

> **Note**
>
> ポート 443 に接続するクライアントは、Application Layer Protocol Negotiation (ALPN) の TLS 拡張機能を実装するとともに、`x-amzn-http-ca` を `ProtocolName` として `ProtocolNameList` に渡す必要があります。詳細については、「AWS IoT デベロッパーガイド」の「プロトコル」を参照してください。

レスポンス

成功すると、レスポンスヘッダーには HTTP 200 ステータスコードが含まれ、レスポンス本文には検出レスポンスドキュメントが含まれます。

> **Note**
>
> AWS IoT Greengrass V2 では AWS IoT Greengrass V1 と同じ検出 API が使用されるため、レスポンスでは Greengrass グループなどの AWS IoT Greengrass V1 概念に従って情報が整理されます。レスポンスには、Greengrass グループのリストが含まれています。AWS IoT Greengrass V2 では、各コアデバイスが独自のグループに属しており、グループにはそのコアデバイスとその接続情報のみが含まれます。

検出レスポンスドキュメントの例

次のドキュメントは、1 つの Greengrass コアデバイスに関連付けられているクライアントデバイスに対するレスポンスを示しています。コアデバイスには、1 つのエンドポイントと 1 つの CA 証明書があります。

```
{
```

```
  "GGGroups": [
    {
      "GGGroupId": "greengrassV2-coreDevice-core-device-01-thing-name",
      "Cores": [
        {
          "thingArn": "core-device-01-thing-arn",
          "Connectivity": [
            {
              "id": "core-device-01-connection-id",
              "hostAddress": "core-device-01-address",
              "portNumber": core-device-01-port,
              "metadata": "core-device-01-description"
            }
          ]
        }
      ],
      "CAs": [
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"
      ]
    }
  ]
}
```

次のドキュメントは、2 つのコアデバイスに関連付けられているクライアントデバイスに対するレスポンスを示しています。コアデバイスには、複数のエンドポイントと複数のグループ CA 証明書があります。

```
{
  "GGGroups": [
    {
      "GGGroupId": "greengrassV2-coreDevice-core-device-01-thing-name",
      "Cores": [
        {
          "thingArn": "core-device-01-thing-arn",
          "Connectivity": [
            {
              "id": "core-device-01-connection-id",
              "hostAddress": "core-device-01-address",
              "portNumber": core-device-01-port,
              "metadata": "core-device-01-connection-1-description"
            },
            {
              "id": "core-device-01-connection-id-2",
```

```
              "hostAddress": "core-device-01-address-2",
              "portNumber": core-device-01-port-2,
              "metadata": "core-device-01-connection-2-description"
            }
          ]
        }
      ],
      "CAs": [
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"
      ]
    },
    {
      "GGGroupId": "greengrassV2-coreDevice-core-device-02-thing-name",
      "Cores": [
        {
          "thingArn":"core-device-02-thing-arn",
          "Connectivity" : [
            {
              "id": "core-device-02-connection-id",
              "hostAddress": "core-device-02-address",
              "portNumber": core-device-02-port,
              "metadata": "core-device-02-connection-1-description"
            }
          ]
        }
      ],
      "CAs": [
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"
      ]
    }
  ]
}
```

cURL でディスカバリ API をテストする

cURL をインストールしている場合は、検出 API をテストできます。次の例では、Greengrass ディスカバリ API エンドポイントへのリクエストを認証するクライアントデバイスの証明書を指定しています。

```
curl -i \
  --cert 1a23bc4d56.cert.pem \
  --key 1a23bc4d56.private.key \
  https://greengrass-ats.iot.us-west-2.amazonaws.com:8443/greengrass/discover/
thing/MyClientDevice1
```

> **Note**
>
> -i 引数で HTTP レスポンスヘッダーを出力するように指定します。このオプションを使用すると、エラーの特定に役立ちます。

リクエストが成功すると、コマンドで下記の例のようなレスポンスが出力されます。

```
{
  "GGGroups": [
    {
      "GGGroupId": "greengrassV2-coreDevice-MyGreengrassCore",
      "Cores": [
        {
          "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
          "Connectivity": [
            {
              "Id": "AUTOIP_192.168.1.4_1",
              "HostAddress": "192.168.1.5",
              "PortNumber": 8883,
              "Metadata": ""
            }
          ]
        }
      ],
      "CAs": [
        "-----BEGIN CERTIFICATE-----\ncert-contents\n-----END CERTIFICATE-----\n"
      ]
    }
  ]
}
```

コマンドでエラーが出力される場合は、「Greengrass ディスカバリの問題のトラブルシューティング」を参照してください。

クライアントデバイスと AWS IoT Core の間の MQTT メッセージのリレー

クライアントデバイスと AWS IoT Core の間で MQTT メッセージや他のデータをリレーできます。クライアントデバイスは、コアデバイスに実行する MQTT ブローカーコンポーネントに接続します。デフォルトでは、コアデバイスはクライアントデバイスと AWS IoT Core の間で MQTT メッセージまたはデータをリレーしません。クライアントデバイスは、デフォルトで MQTT を介して相互にのみ通信できます。

クライアントデバイスと AWS IoT Core の間で MQTT メッセージをリレーするには、MQTT ブリッジコンポーネントが次の手順を実行するように設定します。

- メッセージをクライアントデバイスから AWS IoT Core にリレーします。
- メッセージを AWS IoT Core からクライアントデバイスにリレーします。

> **Note**
>
> MQTT ブリッジは、クライアントデバイスが QoS 0 を使用してローカル MQTT ブローカーをパブリッシュとサブスクライブする場合でも、QoS 1 を使用して AWS IoT Core にパブリッシュとサブスクライブします。その結果、ローカル MQTT ブローカーのクライアントデバイスから AWS IoT Core に MQTT メッセージをリレーするとき、さらにレイテンシーが観察されることがあります。コアデバイスにおける MQTT 設定の詳細については、「MQTT タイムアウトとキャッシュ設定を設定する」を参照してください。

トピック

- MQTT ブリッジコンポーネントの設定とデプロイ
- MQTT メッセージのリレー

MQTT ブリッジコンポーネントの設定とデプロイ

MQTT ブリッジコンポーネントは、それぞれメッセージソースとメッセージの送信先を指定するトピックマッピングのリストを使用します。クライアントデバイスと AWS IoT Core の間でメッセージをリレーするには、MQTT ブリッジコンポーネントをデプロイして、コンポーネント設定で各ソースと宛先トピックを指定します。

MQTT ブリッジコンポーネントをコアデバイスまたはコアデバイスのグループにデプロイするには、aws.greengrass.clientdevices.mqtt.Bridge コンポーネントを含むデプロイを作成します。デプロイの MQTT ブリッジコンポーネント設定で、トピックマッピング mqttTopicMapping を指定します。

次の例では、クライアントデバイスから AWS IoT Core への clients/+/hello/world トピックフィルターに一致するトピックに関するメッセージをリレーするように、MQTT ブリッジコンポーネントを設定するデプロイを定義します。merge 設定の更新には、シリアル化された JSON オブジェクトが必要です。詳細については、「コンポーネント設定の更新」を参照してください。

Console

```
{
  "mqttTopicMapping": {
    "HelloWorldIotCore": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

AWS CLI

```
{
  "components": {
    "aws.greengrass.clientdevices.mqtt.Bridge": {
      "version": "2.0.0",
      "configurationUpdate": {
        "merge": "{\"mqttTopicMapping\":{\"HelloWorldIotCore\":{\"topic\":\"clients/
+/hello/world\",\"source\":\"LocalMqtt\",\"target\":\"IotCore\"}}}"
      }
    }
    ...
  }
}
```

MQTT メッセージのリレー

クライアントデバイスと AWS IoT Core の間で MQTT メッセージをリレーするには、MQTT ブリッジコンポーネントを設定とデプロイして、リレーするトピックを指定します。

Example 例: クライアントデバイスから AWS IoT Core へのトピックに関するメッセージをリレー

次の MQTT ブリッジコンポーネント設定では、クライアントデバイスから AWS IoT Core への `clients/+/hello/world/event` トピックフィルターに一致するトピックに関するメッセージのリレーを指定します。

```
{
  "mqttTopicMapping": {
    "HelloWorldEvent": {
      "topic": "clients/+/hello/world/event",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

Example 例: AWS IoT Core からクライアントデバイスへのトピックに関するメッセージをリレー

次の MQTT ブリッジコンポーネント設定では、AWS IoT Core からクライアントデバイスへの `clients/+/hello/world/event/response` トピックフィルターに一致するトピックに関するメッセージのリレーを指定します。

```
{
  "mqttTopicMapping": {
    "HelloWorldEventConfirmation": {
      "topic": "clients/+/hello/world/event/response",
      "source": "IotCore",
      "target": "LocalMqtt"
    }
  }
}
```

コンポーネント内のクライアントデバイスとやり取りする

コアデバイスに接続されたクライアントデバイスとやり取りするカスタム Greengrass コンポーネントを作成できます。たとえば以下を実行するコンポーネントを作成できます。

- クライアントデバイスからの MQTT メッセージを処理し、AWS クラウド の送信先にデータを送信します。
- MQTT メッセージをクライアントデバイスに送信して、アクションを開始します。

クライアントデバイスは、コアデバイス上で動作する MQTT ブローカーコンポーネントを経由して、コアデバイスへの接続と通信を行います。デフォルトでは、クライアントデバイスは MQTT を介してのみ相互に通信でき、Greengrass コンポーネントはこれらの MQTT メッセージの受信や、クライアントデバイスへのメッセージの送信を行うことはできません。

Greengrass コンポーネントはコアデバイスで通信を行うために、ローカルパブリッシュ/サブスクライブインターフェイスを使用します。Greengrass コンポーネントでクライアントデバイスと通信するには、MQTT ブリッジコンポーネントを設定して以下を実行します。

- クライアントデバイスからローカルのパブリッシュ/サブスクライブへ MQTT メッセージをリレーします。
- ローカルのパブリッシュ/サブスクライブからクライアントデバイスへ MQTT メッセージをリレーします。

Greengrass コンポーネント内のデバイスシャドウとやり取りすることもできます。詳細については、「クライアントデバイスシャドウとやり取りして同期する」を参照してください。

トピック

- MQTT ブリッジコンポーネントの設定とデプロイ
- クライアントデバイスから MQTT メッセージを受信する
- MQTT メッセージをクライアントデバイスに送信する

MQTT ブリッジコンポーネントの設定とデプロイ

MQTT ブリッジコンポーネントは、それぞれメッセージソースとメッセージの送信先を指定するトピックマッピングのリストを使用します。クライアントデバイスと通信するには、MQTT ブリッジコンポーネントをデプロイし、コンポーネント設定で各送信元および送信先のトピックを指定します。

MQTT ブリッジコンポーネントをコアデバイスまたはコアデバイスのグループにデプロイするには、`aws.greengrass.clientdevices.mqtt.Bridge` コンポーネントを含むデプロ

イを作成します。デプロイの MQTT ブリッジコンポーネント設定で、トピックマッピング mqttTopicMapping を指定します。

以下の例は、クライアントデバイスからローカルのパブリッシュ/サブスクライブブローカーに clients/MyClientDevice1/hello/world トピックをリレーするように、MQTT ブリッジコンポーネントを設定するデプロイを定義するものです。merge 設定の更新には、シリアル化された JSON オブジェクトが必要です。詳細については、「コンポーネント設定の更新」を参照してください。

Console

```
{
  "mqttTopicMapping": {
    "HelloWorldPubsub": {
      "topic": "clients/MyClientDevice1/hello/world",
      "source": "LocalMqtt",
      "target": "Pubsub"
    }
  }
}
```

AWS CLI

```
{
  "components": {
    "aws.greengrass.clientdevices.mqtt.Bridge": {
      "version": "2.0.0",
      "configurationUpdate": {
        "merge": "\"mqttTopicMapping\":{\"HelloWorldPubsub\":{\"topic\":\"clients/
MyClientDevice1/hello/world\",\"source\":\"LocalMqtt\",\"target\":\"Pubsub\"}}}"
      }
    }
    ...
  }
}
```

MQTT トピックワイルドカードを使用すると、トピックフィルターに一致するトピックでメッセージをリレーできます。MQTT ブリッジ v2.2.0 以降を使用していて、ソースブローカーがローカルのパブリッシュ/サブスクライブの場合、MQTT トピックワイルドカードをトピックフィルターで使用できます。詳細については、「MQTT ブリッジコンポーネントの設定」を参照してください。

クライアントデバイスから MQTT メッセージを受信する

MQTT Bridge コンポーネントがクライアントデバイスからメッセージを受信するように設定した、ローカルのパブリッシュ/サブスクライブトピックにサブスクライブできます。

カスタムコンポーネントのクライアントデバイスから MQTT メッセージを受信するには

1. MQTT ブリッジコンポーネントの設定とデプロイを行い、クライアントデバイスが発行を行う MQTT トピックから、ローカルのパブリッシュ/サブスクライブトピックにメッセージをリレーします。
2. ローカルパブリッシュ/サブスクライブ IPC インターフェイスを使用して、MQTT ブリッジがメッセージをリレーするトピックにサブスクライブします。詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする と SubscribeToTopic」を参照してください。

クライアントデバイスの接続とテストのチュートリアルには、クライアントデバイスからのメッセージにサブスクライブするコンポーネントを作成するセクションがあります。詳細については、「ステップ 4: クライアントデバイスと通信するコンポーネントを開発する」を参照してください。

MQTT メッセージをクライアントデバイスに送信する

MQTT Bridge コンポーネントがクライアントデバイスにメッセージを送信するように設定した、ローカルのパブリッシュ/サブスクライブトピックに発行できます。

カスタムコンポーネントのクライアントデバイスに MQTT メッセージを発行するには

1. MQTT ブリッジコンポーネントの設定とデプロイを行い、ローカルのパブリッシュ/サブスクライブトピックから、クライアントデバイスがサブスクライブを行う MQTT トピックにメッセージをリレーします。
2. ローカルパブリッシュ/サブスクライブ IPC インターフェイスを使用して、MQTT ブリッジがメッセージをリレーするトピックに発行します。詳細については、「ローカルメッセージをパブリッシュ/サブスクライブする」および「PublishToTopic」を参照してください。

クライアントデバイスシャドウとやり取りして同期する

シャドウマネージャーコンポーネントを使用して、クライアントデバイスシャドウを含むローカルシャドウを管理できます。シャドウマネージャーを使用して次の操作ができます。

- Greengrass コンポーネント内のクライアントデバイスシャドウとやり取りします。

- クライアントデバイスシャドウを AWS IoT Core と同期させる。

> **Note**
>
> デフォルトでは、シャドウマネージャーコンポーネントはシャドウを AWS IoT Core と同期させません。シャドウマネージャーコンポーネントを設定して、同期させるクライアントデバイスシャドウを指定する必要があります。

トピック

前提条件

クライアントデバイスシャドウとやり取りし、クライアントデバイスシャドウを AWS IoT Core と同期させるには、コアデバイスは次の要件を満たしている必要があります。

- コアデバイスでは、クライアントデバイスサポート用の Greengrass コンポーネントに加えて、次のコンポーネントを実行する必要があります。
  - Greengrass nucleus v2.6.0 以降
  - シャドウマネージャー v2.2.0 以降
  - MQTT ブリッジ v2.2.0 以降
- クライアントデバイス認証コンポーネントは、デバイスシャドウトピックでクライアントデバイスが通信できるように設定する必要があります。

シャドウマネージャーがクライアントデバイスと通信できるようにする

デフォルトでは、シャドウマネージャーコンポーネントはクライアントデバイスシャドウを管理しません。この機能を有効にするには、クライアントデバイスとシャドウマネージャーコンポーネント間で MQTT メッセージをリレーする必要があります。クライアントデバイスは MQTT メッセージを使用してデバイスシャドウアップデートを送受信します。シャドウマネージャーコンポーネント

はローカルの Greengrass パブリッシュ/サブスクライブインターフェイスをサブスクライブするので、MQTT ブリッジコンポーネントを設定してデバイスシャドウトピックで MQTT メッセージをリレーできます。

MQTT ブリッジコンポーネントは、それぞれメッセージソースとメッセージの送信先を指定するトピックマッピングのリストを使用します。シャドウマネージャーコンポーネントがクライアントデバイスシャドウを管理できるようにするには、MQTT ブリッジコンポーネントをデプロイし、クライアントデバイスシャドウのシャドウトピックを指定します。ローカル MQTT とローカルのパブリッシュ/サブスクライブの間でメッセージを両方向にリレーするようにブリッジを設定する必要があります。

MQTT ブリッジコンポーネントをコアデバイスまたはコアデバイスのグループにデプロイするには、`aws.greengrass.clientdevices.mqtt.Bridge` コンポーネントを含むデプロイを作成します。デプロイの MQTT ブリッジコンポーネント設定で、トピックマッピング `mqttTopicMapping` を指定します。

クライアントデバイスとシャドウマネージャーコンポーネント間の通信を有効にするように MQTT ブリッジコンポーネントを設定するには、次の例を使用します。

> **Note**
>
> これらの設定例は、AWS IoT Greengrass コンソールで使用できます。AWS IoT Greengrass API を使用する場合、`merge` 設定の更新には、シリアル化された JSON オブジェクトが必要です。そのため、次の JSON オブジェクトをシリアル化して文字列にする必要があります。詳細については、「コンポーネント設定の更新」を参照してください。

Example 例: すべてのクライアントデバイスシャドウを管理する

次の MQTT ブリッジ設定例では、シャドウマネージャーがすべてのクライアントデバイスのすべてのシャドウを管理できるようになります。

```
{
  "mqttTopicMapping": {
    "ShadowsLocalMqttToPubsub": {
      "topic": "$aws/things/+/shadow/#",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
```

```
    "ShadowsPubsubToLocalMqtt": {
      "topic": "$aws/things/+/shadow/#",
      "source": "Pubsub",
      "target": "LocalMqtt"
    }
  }
}
```

Example 例: クライアントデバイスのシャドウを管理する

次の MQTT ブリッジ設定例では、シャドウマネージャーが MyClientDevice という名前のクライアントデバイスのすべてのシャドウを管理できるようになります。

```
{
  "mqttTopicMapping": {
    "ShadowsLocalMqttToPubsub": {
      "topic": "$aws/things/MyClientDevice/shadow/#",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ShadowsPubsubToLocalMqtt": {
      "topic": "$aws/things/MyClientDevice/shadow/#",
      "source": "Pubsub",
      "target": "LocalMqtt"
    }
  }
}
```

Example 例: すべてのクライアントデバイスの名前で指定されたシャドウを管理する

次の MQTT ブリッジ設定例では、シャドウマネージャーがすべてのクライアントデバイスの DeviceConfiguration という名前のシャドウを管理できるようになります。

```
{
  "mqttTopicMapping": {
    "ShadowsLocalMqttToPubsub": {
      "topic": "$aws/things/+/shadow/name/DeviceConfiguration/#",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ShadowsPubsubToLocalMqtt": {
      "topic": "$aws/things/+/shadow/name/DeviceConfiguration/#",
```

```
      "source": "Pubsub",
      "target": "LocalMqtt"
    }
  }
}
```

Example 例: すべてのクライアントデバイスの名前のないシャドウを管理する

次の MQTT ブリッジ設定例では、シャドウマネージャーがすべてのクライアントデバイスの名前のないシャドウを管理できるようになりますが、名前付きシャドウは管理できません。

```
{
  "mqttTopicMapping": {
    "DeleteShadowLocalMqttToPubsub": {
      "topic": "$aws/things/+/shadow/delete",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "DeleteShadowPubsubToLocalMqtt": {
      "topic": "$aws/things/+/shadow/delete/#",
      "source": "Pubsub",
      "target": "LocalMqtt"
    },
    "GetShadowLocalMqttToPubsub": {
      "topic": "$aws/things/+/shadow/get",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "GetShadowPubsubToLocalMqtt": {
      "topic": "$aws/things/+/shadow/get/#",
      "source": "Pubsub",
      "target": "LocalMqtt"
    },
    "UpdateShadowLocalMqttToPubsub": {
      "topic": "$aws/things/+/shadow/update",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "UpdateShadowPubsubToLocalMqtt": {
      "topic": "$aws/things/+/shadow/update/#",
      "source": "Pubsub",
      "target": "LocalMqtt"
    }
```

```
  }
}
```

コンポーネント内のクライアントデバイスシャドウとやり取りする

ローカルシャドウサービスを使用するカスタムコンポーネントを開発して、クライアントデバイスのローカルシャドウドキュメントの読み取りと変更ができます。詳細については、「コンポーネントのシャドウとやり取りする」を参照してください。

クライアントデバイスシャドウを AWS IoT Core と同期させる

シャドウマネージャーコンポーネントを設定して、ローカルクライアントデバイスシャドウ状態を AWS IoT Core と同期させることができます。詳細については、「ローカルデバイスシャドウを AWS IoT Core と同期する」を参照してください。

クライアントデバイスのトラブルシューティング

このセクションのトラブルシューティング情報と解決策は、Greengrass クライアントデバイスとクライアントデバイスのコンポーネントとの問題解決に役立ちます。

トピック

- Greengrass 検出の問題
- MQTT 接続の問題

Greengrass 検出の問題

次の情報を使用して、Greengrass 検出に関する問題のトラブルシューティングを行います。これらの問題は、クライアントデバイスが [Greengrass discovery API] (Greengrass 検出 API) に接続できる Greengrass コアデバイスを識別します。

トピック

- Greengrass 検出の問題 (HTTP API)
- Greengrass 検出の問題 (AWS IoT Device SDK v2 Python)
- Greengrass 検出の問題 (AWS IoT Device SDK v2 for C++)
- Greengrass 検出の問題 (AWS IoT Device SDK v2 for JavaScript)

- Greengrass 検出の問題 (AWS IoT Device SDK v2 for Java)

Greengrass 検出の問題 (HTTP API)

次の情報を使用して、Greengrass 検出に関する問題のトラブルシューティングを行います。[test the discovery API with cURL] (cURL でディスカバリー API をテストする) と、これらのエラーが表示される場合があります。

トピック

- curl: (52) Empty reply from server
- HTTP 403: {"message":null,"traceId":"a1b2c3d4-5678-90ab-cdef-11111EXAMPLE"}
- HTTP 404: {"errorMessage":"The thing provided for discovery was not found"}

curl: (52) Empty reply from server

リクエストで非アクティブな AWS IoT 証明書を指定すると、このエラーが表示される場合があります。

クライアントデバイスに証明書が添付されていること、および証明書がアクティブであることを確認します。詳細については、「AWS IoT Coreデベロッパーガイド」の「クライアント証明書にモノまたはポリシーをアタッチする」および「クライアント証明書をアクティブ化または非アクティブ化する」を参照してください。

HTTP 403: {"message":null,"traceId":"a1b2c3d4-5678-90ab-cdef-11111EXAMPLE"}

クライアントデバイスに `greengrass:Discover` を呼び出すアクセス許可がない場合は、このエラーが表示されることがあります。

クライアントデバイスの証明書に、`greengrass:Discover` を許可するポリシーがあることを確認します。このアクセス許可には、`Resource` セクションの [thing policy variables] (モノのポリシー変数) (`iot:Connection.Thing.*`) を使用できません。詳細については、「検出認証と認可」を参照してください。

HTTP 404: {"errorMessage":"The thing provided for discovery was not found"}

次の場合にこのエラーが発生する可能性があります。

- クライアントデバイスが Greengrass コアデバイスまたは AWS IoT Greengrass V1 グループに関連付けられていません。

- クライアントデバイスに関連付けられている Greengrass コアデバイスまたは AWS IoT Greengrass V1 グループには、MQTT ブローカーエンドポイントがありません。
- クライアントデバイスに関連付けられている Greengrass コアデバイスには、クライアントデバイス認証コンポーネントを実行しているものがありません。

クライアントデバイスが、接続するコアデバイスに関連付けられていることを確認します。次に、コアデバイスがクライアントデバイス認証コンポーネントを実行していて、コアデバイスに少なくとも 1 つの MQTT ブローカーエンドポイントがあることを確認します。詳細については、次を参照してください。

- クライアントデバイスを関連付ける
- コアデバイスのエンドポイントを管理
- クラウドディスカバリを設定する (コンソール)

Greengrass 検出の問題 (AWS IoT Device SDK v2 Python)

次の情報を使用して、AWS IoT Device SDK v2 for Python での Greengrass 検出に関する問題のトラブルシューティングを行います。

トピック

- awscrt.exceptions.AwsCrtError: AWS_ERROR_HTTP_CONNECTION_CLOSED: The connection has closed or is closing.
- awsiot.greengrass_discovery.DiscoveryException: ('Error during discover call: response_code=403', 403)
- awsiot.greengrass_discovery.DiscoveryException: ('Error during discover call: response_code=404', 404)

awscrt.exceptions.AwsCrtError: AWS_ERROR_HTTP_CONNECTION_CLOSED: The connection has closed or is closing.

リクエストで非アクティブな AWS IoT 証明書を指定すると、このエラーが表示される場合があります。

クライアントデバイスに証明書が添付されていること、および証明書がアクティブであることを確認します。詳細については、「AWS IoT Coreデベロッパーガイド」の「クライアント証明書にモノま

たはポリシーをアタッチする」および「クライアント証明書をアクティブ化または非アクティブ化する」を参照してください。

awsiot.greengrass_discovery.DiscoveryException: ('Error during discover call: response_code=403', 403)

クライアントデバイスに `greengrass:Discover` を呼び出すアクセス許可がない場合は、このエラーが表示されることがあります。

クライアントデバイスの証明書に、`greengrass:Discover` を許可するポリシーがあることを確認します。このアクセス許可には、`Resource` セクションの [thing policy variables] (モノのポリシー変数) (`iot:Connection.Thing.*`) を使用できません。詳細については、「検出認証と認可」を参照してください。

awsiot.greengrass_discovery.DiscoveryException: ('Error during discover call: response_code=404', 404)

次の場合にこのエラーが発生する可能性があります。

- クライアントデバイスが Greengrass コアデバイスまたは AWS IoT Greengrass V1 グループに関連付けられていません。
- クライアントデバイスに関連付けられている Greengrass コアデバイスまたは AWS IoT Greengrass V1 グループには、MQTT ブローカーエンドポイントがありません。
- クライアントデバイスに関連付けられている Greengrass コアデバイスには、クライアントデバイス認証コンポーネントを実行しているものがありません。

クライアントデバイスが、接続するコアデバイスに関連付けられていることを確認します。次に、コアデバイスがクライアントデバイス認証コンポーネントを実行していて、コアデバイスに少なくとも 1 つの MQTT ブローカーエンドポイントがあることを確認します。詳細については、次を参照してください。

- クライアントデバイスを関連付ける
- コアデバイスのエンドポイントを管理
- クラウドディスカバリを設定する (コンソール)

Greengrass 検出の問題 (AWS IoT Device SDK v2 for C++)

次の情報を使用して、AWS IoT Device SDK v2 for C++ での Greengrass 検出に関する問題のトラブルシューティングを行います。

トピック

aws-c-http: AWS_ERROR_HTTP_CONNECTION_CLOSED, The connection has closed or is closing.

リクエストで非アクティブな AWS IoT 証明書を指定すると、このエラーが表示される場合があります。

クライアントデバイスに証明書が添付されていること、および証明書がアクティブであることを確認します。詳細については、「AWS IoT Coreデベロッパーガイド」の「クライアント証明書にモノまたはポリシーをアタッチする」および「クライアント証明書をアクティブ化または非アクティブ化する」を参照してください。

aws-c-common: AWS_ERROR_UNKNOWN, Unknown error. (HTTP 403)

クライアントデバイスに `greengrass:Discover` を呼び出すアクセス許可がない場合は、このエラーが表示されることがあります。

クライアントデバイスの証明書に、`greengrass:Discover` を許可するポリシーがあることを確認します。このアクセス許可には、`Resource` セクションの [thing policy variables] (モノのポリシー変数) (`iot:Connection.Thing.*`) を使用できません。詳細については、「検出認証と認可」を参照してください。

aws-c-common: AWS_ERROR_UNKNOWN, Unknown error. (HTTP 404)

次の場合にこのエラーが発生する可能性があります。

- クライアントデバイスが Greengrass コアデバイスまたは AWS IoT Greengrass V1 グループに関連付けられていません。
- クライアントデバイスに関連付けられている Greengrass コアデバイスまたは AWS IoT Greengrass V1 グループには、MQTT ブローカーエンドポイントがありません。

- クライアントデバイスに関連付けられている Greengrass コアデバイスには、クライアントデバイス認証コンポーネントを実行しているものがありません。

クライアントデバイスが、接続するコアデバイスに関連付けられていることを確認します。次に、コアデバイスがクライアントデバイス認証コンポーネントを実行していて、コアデバイスに少なくとも 1 つの MQTT ブローカーエンドポイントがあることを確認します。詳細については、次を参照してください。

- クライアントデバイスを関連付ける
- コアデバイスのエンドポイントを管理
- クラウドディスカバリを設定する (コンソール)

Greengrass 検出の問題 (AWS IoT Device SDK v2 for JavaScript)

次の情報を使用して、 AWS IoT Device SDKv2 for での Greengrass 検出に関する問題のトラブルシューティングを行います JavaScript。

トピック

- Error: aws-c-http: AWS_ERROR_HTTP_CONNECTION_CLOSED, The connection has closed or is closing.
- Error: Discovery failed (headers: [object Object]) { response_code: 403 }
- Error: Discovery failed (headers: [object Object]) { response_code: 404 }
- Error: Discovery failed (headers: [object Object])

Error: aws-c-http: AWS_ERROR_HTTP_CONNECTION_CLOSED, The connection has closed or is closing.

リクエストで非アクティブな AWS IoT 証明書を指定すると、このエラーが表示される場合があります。

クライアントデバイスに証明書が添付されていること、および証明書がアクティブであることを確認します。詳細については、「AWS IoT Coreデベロッパーガイド」の「クライアント証明書にモノまたはポリシーをアタッチする」および「クライアント証明書をアクティブ化または非アクティブ化する」を参照してください。

Error: Discovery failed (headers: [object Object]) { response_code: 403 }

クライアントデバイスに `greengrass:Discover` を呼び出すアクセス許可がない場合は、このエラーが表示されることがあります。

クライアントデバイスの証明書に、`greengrass:Discover` を許可するポリシーがあることを確認します。このアクセス許可には、`Resource` セクションの [thing policy variables] (モノのポリシー変数) (`iot:Connection.Thing.*`) を使用できません。詳細については、「検出認証と認可」を参照してください。

Error: Discovery failed (headers: [object Object]) { response_code: 404 }

次の場合にこのエラーが発生する可能性があります。

- クライアントデバイスが Greengrass コアデバイスまたは AWS IoT Greengrass V1 グループに関連付けられていません。
- クライアントデバイスに関連付けられている Greengrass コアデバイスまたは AWS IoT Greengrass V1 グループには、MQTT ブローカーエンドポイントがありません。
- クライアントデバイスに関連付けられている Greengrass コアデバイスには、クライアントデバイス認証コンポーネントを実行しているものがありません。

クライアントデバイスが、接続するコアデバイスに関連付けられていることを確認します。次に、コアデバイスがクライアントデバイス認証コンポーネントを実行していて、コアデバイスに少なくとも 1 つの MQTT ブローカーエンドポイントがあることを確認します。詳細については、次を参照してください。

- クライアントデバイスを関連付ける
- コアデバイスのエンドポイントを管理
- クラウドディスカバリを設定する (コンソール)

Error: Discovery failed (headers: [object Object])

Greengrass 検出サンプルを実行すると、このエラー (HTTP レスポンスコードなし) が表示される場合があります。このエラーは、さまざまな理由で発生する可能性があります。

- クライアントデバイスに `greengrass:Discover` を呼び出すアクセス許可がない場合は、このエラーが表示されることがあります。

クライアントデバイスの証明書に、`greengrass:Discover` を許可するポリシーがあることを確認します。このアクセス許可には、`Resource` セクションの [thing policy variables] (モノのポリシー変数) (`iot:Connection.Thing.*`) を使用できません。詳細については、「検出認証と認可」を参照してください。

- 次の場合にこのエラーが発生する可能性があります。
  - クライアントデバイスが Greengrass コアデバイスまたは AWS IoT Greengrass V1 グループに関連付けられていません。
  - クライアントデバイスに関連付けられている Greengrass コアデバイスまたは AWS IoT Greengrass V1 グループには、MQTT ブローカーエンドポイントがありません。
  - クライアントデバイスに関連付けられている Greengrass コアデバイスには、クライアントデバイス認証コンポーネントを実行しているものがありません。

  クライアントデバイスが、接続するコアデバイスに関連付けられていることを確認します。次に、コアデバイスがクライアントデバイス認証コンポーネントを実行していて、コアデバイスに少なくとも 1 つの MQTT ブローカーエンドポイントがあることを確認します。詳細については、次を参照してください。
  - クライアントデバイスを関連付ける
  - コアデバイスのエンドポイントを管理
  - クラウドディスカバリを設定する (コンソール)

Greengrass 検出の問題 (AWS IoT Device SDK v2 for Java)

次の情報を使用して、AWS IoT Device SDK v2 for Java での Greengrass 検出に関する問題のトラブルシューティングを行います。

トピック

- software.amazon.awssdk.crt.CrtRuntimeException: Error Getting Response Status Code from HttpStream. (aws_last_error: AWS_ERROR_HTTP_DATA_NOT_AVAILABLE(2062), This data is not yet available.)
- java.lang.RuntimeException: Error x-amzn-ErrorType(403)
- java.lang.RuntimeException: Error x-amzn-ErrorType(404)

software.amazon.awssdk.crt.CrtRuntimeException: Error Getting Response Status Code from HttpStream. (aws_last_error: AWS_ERROR_HTTP_DATA_NOT_AVAILABLE(2062), This data is not yet available.)

リクエストで非アクティブな AWS IoT 証明書を指定すると、このエラーが表示される場合があります。

クライアントデバイスに証明書が添付されていること、および証明書がアクティブであることを確認します。詳細については、「AWS IoT Coreデベロッパーガイド」の「クライアント証明書にモノまたはポリシーをアタッチする」および「クライアント証明書をアクティブ化または非アクティブ化する」を参照してください。

java.lang.RuntimeException: Error x-amzn-ErrorType(403)

クライアントデバイスに `greengrass:Discover` を呼び出すアクセス許可がない場合は、このエラーが表示されることがあります。

クライアントデバイスの証明書に、`greengrass:Discover` を許可するポリシーがあることを確認します。このアクセス許可には、`Resource` セクションの [thing policy variables] (モノのポリシー変数) (`iot:Connection.Thing.*`) を使用できません。詳細については、「検出認証と認可」を参照してください。

java.lang.RuntimeException: Error x-amzn-ErrorType(404)

次の場合にこのエラーが発生する可能性があります。

- クライアントデバイスが Greengrass コアデバイスまたは AWS IoT Greengrass V1 グループに関連付けられていません。
- クライアントデバイスに関連付けられている Greengrass コアデバイスまたは AWS IoT Greengrass V1 グループには、MQTT ブローカーエンドポイントがありません。
- クライアントデバイスに関連付けられている Greengrass コアデバイスには、クライアントデバイス認証コンポーネントを実行しているものがありません。

クライアントデバイスが、接続するコアデバイスに関連付けられていることを確認します。次に、コアデバイスがクライアントデバイス認証コンポーネントを実行していて、コアデバイスに少なくとも 1 つの MQTT ブローカーエンドポイントがあることを確認します。詳細については、次を参照してください。

- クライアントデバイスを関連付ける

- コアデバイスのエンドポイントを管理
- クラウドディスカバリを設定する (コンソール)

MQTT 接続の問題

クライアントデバイス MQTT 接続の問題のトラブルシューティングには、次の情報を使用します。これらの問題は、クライアントデバイスが MQTT 経由でコアデバイスに接続しようとする際に発生する場合があります。

トピック

- io.moquette.broker.Authorizator: Client does not have read permissions on the topic
- MQTT 接続の問題 (Python)
- MQTT 接続の問題 (C++)
- MQTT 接続の問題 (Java)
- MQTT 接続の問題 (JavaScript)

io.moquette.broker.Authorizator: Client does not have read permissions on the topic

このエラーは、クライアントデバイスがアクセス許可を持たない MQTT トピックをサブスクライブしようとしたときに、Greengrass ログに表示されることがあります。エラーメッセージにはトピックが含まれています。

クライアントデバイス認証コンポーネントの設定に、次の項目が含まれていることを確認します。

- クライアントデバイスと一致するデバイスグループ。
- トピックの `mqtt:subscribe` アクセス許可を付与する、そのデバイスグループのクライアントデバイス承認ポリシー。

クライアントデバイス認証コンポーネントをデプロイおよび設定する方法の詳細については、次を参照してください。

- クラウドディスカバリを設定する (コンソール)
- クライアントデバイス認証
- デプロイの作成

MQTT 接続の問題 (Python)

次の情報を使用して、AWS IoT Device SDK v2 for Python 使用時のクライアントデバイスの MQTT 接続に関する問題のトラブルシューティングを行います。

トピック

AWS_ERROR_MQTT_PROTOCOL_ERROR: Protocol error occurred

クライアントデバイス認証コンポーネントで、接続するアクセス許可をクライアントデバイスに付与するクライアントデバイス承認ポリシーが定義されていない場合にこのエラーが発生する可能性があります。

クライアントデバイス認証コンポーネントの設定に次の項目が含まれていることを確認します。

- クライアントデバイスと一致するデバイスグループ。
- クライアントデバイスの `mqtt:connect` アクセス許可を付与する、そのデバイスグループのクライアントデバイス承認ポリシー。

クライアントデバイス認証コンポーネントをデプロイおよび設定する方法の詳細については、次を参照してください。

- クラウドディスカバリを設定する (コンソール)
- クライアントデバイス認証
- デプロイの作成

AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

クライアントデバイス認証コンポーネントで、接続するアクセス許可をクライアントデバイスに付与するクライアントデバイス承認ポリシーが定義されていない場合にこのエラーが発生する可能性があります。

クライアントデバイス認証コンポーネントの設定に次の項目が含まれていることを確認します。

- クライアントデバイスと一致するデバイスグループ。

- クライアントデバイスの `mqtt:connect` アクセス許可を付与する、そのデバイスグループのクライアントデバイス承認ポリシー。

クライアントデバイス認証コンポーネントをデプロイおよび設定する方法の詳細については、次を参照してください。

- クラウドディスカバリを設定する (コンソール)
- クライアントデバイス認証
- デプロイの作成

MQTT 接続の問題 (C++)

次の情報を使用して、AWS IoT Device SDK v2 for C++ 使用時のクライアントデバイス MQTT 接続に関する問題のトラブルシューティングを行います。

トピック

- AWS_ERROR_MQTT_PROTOCOL_ERROR: Protocol error occurred
- AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

AWS_ERROR_MQTT_PROTOCOL_ERROR: Protocol error occurred

クライアントデバイス認証コンポーネントで、接続するアクセス許可をクライアントデバイスに付与するクライアントデバイス承認ポリシーが定義されていない場合にこのエラーが発生する可能性があります。

クライアントデバイス認証コンポーネントの設定に次の項目が含まれていることを確認します。

- クライアントデバイスと一致するデバイスグループ。
- クライアントデバイスの `mqtt:connect` アクセス許可を付与する、そのデバイスグループのクライアントデバイス承認ポリシー。

クライアントデバイス認証コンポーネントをデプロイおよび設定する方法の詳細については、次を参照してください。

- クラウドディスカバリを設定する (コンソール)
- クライアントデバイス認証

- デプロイの作成

AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

クライアントデバイス認証コンポーネントで、接続するアクセス許可をクライアントデバイスに付与するクライアントデバイス承認ポリシーが定義されていない場合にこのエラーが発生する可能性があります。

クライアントデバイス認証コンポーネントの設定に次の項目が含まれていることを確認します。

- クライアントデバイスと一致するデバイスグループ。
- クライアントデバイスの `mqtt:connect` アクセス許可を付与する、そのデバイスグループのクライアントデバイス承認ポリシー。

クライアントデバイス認証コンポーネントをデプロイおよび設定する方法の詳細については、次を参照してください。

- クラウドディスカバリを設定する (コンソール)
- クライアントデバイス認証
- デプロイの作成

MQTT 接続の問題 (Java)

次の情報を使用して、AWS IoT Device SDK v2 for Java 使用時のクライアントデバイス MQTT 接続に関する問題のトラブルシューティングを行います。

トピック

- software.amazon.awssdk.crt.mqtt.MqttException: Protocol error occurred
- AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

software.amazon.awssdk.crt.mqtt.MqttException: Protocol error occurred

クライアントデバイス認証コンポーネントで、接続するアクセス許可をクライアントデバイスに付与するクライアントデバイス承認ポリシーが定義されていない場合にこのエラーが発生する可能性があります。

クライアントデバイス認証コンポーネントの設定に次の項目が含まれていることを確認します。

- クライアントデバイスと一致するデバイスグループ。
- クライアントデバイスの `mqtt:connect` アクセス許可を付与する、そのデバイスグループのクライアントデバイス承認ポリシー。

クライアントデバイス認証コンポーネントをデプロイおよび設定する方法の詳細については、次を参照してください。

- クラウドディスカバリを設定する (コンソール)
- クライアントデバイス認証
- デプロイの作成

AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

クライアントデバイス認証コンポーネントで、接続するアクセス許可をクライアントデバイスに付与するクライアントデバイス承認ポリシーが定義されていない場合にこのエラーが発生する可能性があります。

クライアントデバイス認証コンポーネントの設定に次の項目が含まれていることを確認します。

- クライアントデバイスと一致するデバイスグループ。
- クライアントデバイスの `mqtt:connect` アクセス許可を付与する、そのデバイスグループのクライアントデバイス承認ポリシー。

クライアントデバイス認証コンポーネントをデプロイおよび設定する方法の詳細については、次を参照してください。

- クラウドディスカバリを設定する (コンソール)
- クライアントデバイス認証
- デプロイの作成

MQTT 接続の問題 (JavaScript)

次の情報を使用して、 AWS IoT Device SDK v2 for JavaScriptを使用する場合のクライアントデバイスの MQTT 接続に関する問題のトラブルシューティングを行います。

トピック

- AWS_ERROR_MQTT_PROTOCOL_ERROR: Protocol error occurred
- AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

AWS_ERROR_MQTT_PROTOCOL_ERROR: Protocol error occurred

クライアントデバイス認証コンポーネントで、接続するアクセス許可をクライアントデバイスに付与するクライアントデバイス承認ポリシーが定義されていない場合にこのエラーが発生する可能性があります。

クライアントデバイス認証コンポーネントの設定に次の項目が含まれていることを確認します。

- クライアントデバイスと一致するデバイスグループ。
- クライアントデバイスの `mqtt:connect` アクセス許可を付与する、そのデバイスグループのクライアントデバイス承認ポリシー。

クライアントデバイス認証コンポーネントをデプロイおよび設定する方法の詳細については、次を参照してください。

- クラウドディスカバリを設定する (コンソール)
- クライアントデバイス認証
- デプロイの作成

AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

クライアントデバイス認証コンポーネントで、接続するアクセス許可をクライアントデバイスに付与するクライアントデバイス承認ポリシーが定義されていない場合にこのエラーが発生する可能性があります。

クライアントデバイス認証コンポーネントの設定に次の項目が含まれていることを確認します。

- クライアントデバイスと一致するデバイスグループ。
- クライアントデバイスの `mqtt:connect` アクセス許可を付与する、そのデバイスグループのクライアントデバイス承認ポリシー。

クライアントデバイス認証コンポーネントをデプロイおよび設定する方法の詳細については、次を参照してください。

- クラウドディスカバリを設定する (コンソール)
- クライアントデバイス認証
- デプロイの作成

デバイスシャドウとやり取り

Greengrass コアデバイスはコンポーネントを使用して、AWS IoT デバイスシャドウとやり取りできます。「シャドウ」は、AWS IoT モノの現在または目的の状態に関する情報を保存する JSON ドキュメントです。シャドウは、デバイスが AWS IoT に接続されているかどうかにかかわらず、他の AWS IoT Greengrass コンポーネントでデバイスの状態を利用できるようにします。AWS IoT デバイスは、それぞれ独自に従来の名前なしシャドウを持ちます。また、デバイスごとに名前付きシャドウを複数作成することもできます。

デバイスおよびサービスは、MQTT および予約された MQTT シャドウトピック、Device Shadow REST API を使用する HTTP、AWS CLI for AWS IoT を使用して、クラウドシャドウを作成、更新、削除できます。

シャドウマネージャーコンポーネントを使用すると、Greengrass コンポーネントはローカルシャドウサービスおよびローカルパブリッシュ/サブスクライブシャドウトピックを使用して、ローカルシャドウを作成、更新、削除できるようになります。シャドウマネージャーは、これらのローカルシャドウドキュメントのコアデバイスへの保存を管理し、クラウドシャドウとシャドウの状態情報を同期する処理を行います。

シャドウマネージャーコンポーネントを使用すると、コアデバイスに接続するクライアントデバイスの、ローカルシャドウの管理もできるようになります。シャドウマネージャがクライアントデバイスのシャドウを管理できるようにするには、ローカルの MQTT ブローカーと、ローカルのパブリッシュ/サブスクライブサービスとの間でメッセージをリレーできるように、MQTT ブリッジコンポーネントを設定します。詳細については、「クライアントデバイスシャドウとやり取りして同期する」を参照してください。

AWS IoT デバイスシャドウのコンセプトの詳細については、「AWS IoT デベロッパーガイド」の「AWS IoT デバイスシャドウサービス」を参照してください。

トピック

- コンポーネントのシャドウとやり取りする
- ローカルデバイスシャドウを AWS IoT Core と同期する

コンポーネントのシャドウとやり取りする

ローカルシャドウサービスを使用して、ローカルシャドウドキュメントと、クライアントデバイスシャドウドキュメントの、読み取りと変更を実行する、カスタムコンポーネント (Lambda 関数コンポーネントなど) を開発できます。

カスタムコンポーネントは、AWS IoT Device SDK の AWS IoT Greengrass Core IPC ライブラリを使用して、ローカルシャドウサービスとやり取りを行います。シャドウマネージャーコンポーネントを使用すると、コアデバイスでローカルシャドウサービスを有効にできます。

Greengrass コアデバイスにシャドウマネージャーコンポーネントをデプロイするには、`aws.greengrass.ShadowManager` コンポーネントを含むデプロイを作成します。

> **Note**
>
> デフォルトでは、シャドウマネージャーコンポーネントをデプロイすると、ローカルシャドウのオペレーションのみが有効になります。AWS IoT Greengrass が、コアデバイスのシャドウまたはクライアントデバイスのシャドウのシャドウ状態情報を、AWS IoT Core の対応するクラウドシャドウドキュメントに同期できるようにするには、`synchronize` パラメータを含むシャドウマネージャーコンポーネントの設定更新を作成する必要があります。詳細については、「ローカルデバイスシャドウを AWS IoT Core と同期する」を参照してください。

トピック

シャドウの状態の取得と変更

シャドウ IPC 操作は、ローカルシャドウドキュメントの状態情報を取得および更新します。シャドウマネージャーコンポーネントは、これらのシャドウドキュメントのコアデバイスへの保存処理を行います。

ローカルシャドウの状態を変更するには

1. カスタムコンポーネントのレシピに承認ポリシーを追加して、コンポーネントがローカルシャドートピックに関するメッセージを受信できるようにします。

承認ポリシーの例は、「Local shadow IPC authorization policy examples」(ローカルシャドウ IPC 承認ポリシーの例) を参照してください。

2. シャドウ IPC オペレーションを使用して、シャドウの状態情報を取得および変更します。コンポーネントコードでシャドウ IPC オペレーションを使用する方法については、「ローカルシャドウとやり取り」を参照してください。

> **Note**
>
> コアデバイスがクライアントデバイスシャドウとやり取りできるようにするには、MQTT ブリッジコンポーネントを設定してデプロイする必要もあります。詳細については、「Enable shadow manager to communicate with client devices」(シャドウマネージャーがクライアントデバイスと通信できるようにする) を参照してください。

シャドウの状態の変化に対応する

Greengrass コンポーネントはコアデバイスで通信を行うために、ローカルパブリッシュ/サブスクライブインターフェイスを使用します。カスタムコンポーネントがシャドウの状態の変更に対応できるようにするには、ローカルのパブリッシュ/サブスクライブトピックにサブスクライブします。これにより、コンポーネントはローカルシャドートピックに関するメッセージを受信し、それらのメッセージを処理できます。

ローカルシャドートピックは、AWS IoT デバイスシャドウ MQTT トピックと同じ形式を使用します。シャドウトピックの詳細については、「AWS IoT デベロッパーガイド」の「デバイスシャドウ MQTT トピック」を参照してください。

ローカルシャドウの状態の変化に対応するには

1. カスタムコンポーネントのレシピにアクセスコントロールポリシーを追加して、コンポーネントがローカルシャドートピックに関するメッセージを受信できるようにします。

   承認ポリシーの例は、「Local shadow IPC authorization policy examples」(ローカルシャドウ IPC 承認ポリシーの例) を参照してください。
2. コンポーネントでカスタムアクションを開始するには、メッセージを受信するシャドートピックにサブスクライブする `SubscribeToTopic` IPC オペレーションを使用します。コンポーネントコードでローカルパブリッシュ/サブスクライブ IPC オペレーションを使用する方法については、「ローカルメッセージをパブリッシュ/サブスクライブする」を参照してください。

3. Lambda 関数を呼び出すには、イベントソース設定を使用してシャドートピックの名前を指定し、それがローカルパブリッシュ/サブスクライブトピックであることを指定します。Lambda 関数コンポーネントの作成方法については、「AWS Lambda 関数を実行する」を参照してください。

> **Note**
>
> コアデバイスがクライアントデバイスシャドウとやり取りできるようにするには、MQTT ブリッジコンポーネントを設定してデプロイする必要もあります。詳細については、「Enable shadow manager to communicate with client devices」(シャドウマネージャーがクライアントデバイスと通信できるようにする) を参照してください。

ローカルデバイスシャドウを AWS IoT Core と同期する

シャドウマネージャーコンポーネントにより、AWS IoT Greengrass はローカルデバイスシャドウ状態を AWS IoT Core と同期できるようになります。シャドウマネージャーコンポーネントの設定を変更して、`synchronization` 設定パラメータを含め、デバイスの AWS IoT のモノの名前と、同期するシャドウを指定する必要があります。

シャドウを同期するようにシャドウマネージャーを設定すると、ローカルシャドウドキュメントとクラウドシャドウドキュメントのどちらで変更が発生するかに関係なく、指定したシャドウのすべての状態変更が同期されます。

また、シャドウマネージャーコンポーネントがシャドウをリアルタイムで同期するか、定期的な間隔で同期するかを指定することもできます。デフォルトでは、シャドウマネージャーコンポーネントはリアルタイムでシャドウを同期するため、コアデバイスは更新が発生するたびに AWS IoT Core との間でシャドウの更新を送受信します。定期的な間隔を設定して、帯域幅の使用量と料金を削減できます。

トピック

前提条件

ローカルシャドウを AWS IoT Core と同期させるには、Greengrass コアデバイスの AWS IoT ポリシーを、次の AWS IoT Core シャドウポリシーアクションを許可するように設定する必要があります。

- `iot:GetThingShadow`
- `iot:UpdateThingShadow`
- `iot:DeleteThingShadow`

詳細については、次を参照してください。

- 「AWS IoT デベロッパーガイド」の「AWS IoT Core ポリシーアクション」
- AWS IoT Greengrass V2 コアデバイス向けの最低限の AWS IoT ポリシー
- コアデバイスの AWS IoT ポリシーを更新する

シャドウマネージャーコンポーネントを設定する

シャドウマネージャーでは、ローカルシャドウドキュメントのシャドウ状態情報を AWS IoT Core のクラウドシャドウドキュメントに同期するには、シャドウ名のマッピングのリストが必要です。

シャドウの状態を同期するには、`aws.greengrass.ShadowManager` コンポーネントを含むデプロイを作成し、デプロイのシャドウマネージャー設定の `synchronize` 設定パラメータで同期するシャドウを指定します。

> **Note**
>
> コアデバイスがクライアントデバイスシャドウとやり取りできるようにするには、MQTT ブリッジコンポーネントを設定してデプロイする必要もあります。詳細については、「Enable shadow manager to communicate with client devices」(シャドウマネージャーがクライアントデバイスと通信できるようにする) を参照してください。

次の設定更新例では、シャドウマネージャーコンポーネントに次のシャドウを AWS IoT Core と同期するように指示します。

- コアデバイスの古典的なシャドウ

- コアデバイスの名前付き MyCoreShadow
- MyDevice2 という名前の IoT モノのクラシックなシャドウ
- MyDevice1 という名前の IoT モノの名前付きシャドウ MyShadowA および MyShadowB

この設定更新では、シャドウを AWS IoT Core とリアルタイムで同期するように指定されています。シャドウマネージャー v2.1.0 以降を使用している場合、定期的な間隔でシャドウを同期させるようにシャドウマネージャーコンポーネントを設定できます。この機能を設定するには、同期ストラテジーを periodic に変更し、間隔の delay を秒単位で指定します。詳細については、「シャドウマネージャーコンポーネント」の「ストラテジー設定パラメータ」を参照してください。

この設定更新では、AWS IoT Core とコアデバイスの間の両方向でシャドウが同期するように指定されています。シャドウマネージャー v2.2.0 以降を使用している場合、シャドウを一方向にのみ同期するようにシャドウマネージャーコンポーネントを設定できます。この機能を設定するには、同期 direction を deviceToCloud または cloudToDevice に変更します。詳細については、シャドウマネージャーコンポーネントの direction 設定パラメータを参照してください。

```
{
  "strategy": {
    "type": "realTime"
  },
  "synchronize": {
    "coreThing": {
      "classic": true,
      "namedShadows": [
        "MyCoreShadow"
      ]
    },
    "shadowDocuments": [
      {
        "thingName": "MyDevice1",
        "classic": false,
        "namedShadows": [
          "MyShadowA",
          "MyShadowB"
        ]
      },
      {
        "thingName": "MyDevice2",
        "classic": true,
        "namedShadows": [ ]
```

```
      }
    ],
    "direction": "betweenDeviceAndCloud"
  }
}
```

ローカルシャドウを同期する

Greengrass コアデバイスが AWS IoT クラウドに接続されている場合、シャドウマネージャーは、コンポーネント設定で指定したシャドウに対して次のタスクを実行します。この動作は、指定するシャドウ同期方向設定オプションによって異なります。デフォルトでは、シャドウマネージャーは betweenDeviceAndCloud オプションを使用して、シャドウを両方向に同期させます。シャドウマネージャー v2.2.0 以降を使用している場合、シャドウを一方向 (cloudToDevice または deviceToCloud) にのみ同期するようにコアデバイスを設定できます。

- シャドウ同期の方向設定が betweenDeviceAndCloud または cloudToDevice の場合、シャドウマネージャーは、AWS IoT Core のクラウドシャドウドキュメントから報告された状態情報を取得します。そして、ローカルに保存されたシャドウドキュメントを更新して、デバイスの状態を同期させます。
- シャドウ同期の方向設定が betweenDeviceAndCloud または deviceToCloud の場合、シャドウマネージャーは、デバイスの現在の状態をクラウドシャドウドキュメントに公開します。

シャドウマージの競合動作

コアデバイスがインターネットから切断されている場合など、場合によっては、シャドウマネージャーが変更を同期する前に、ローカルシャドウサービスと AWS IoT クラウドでシャドウが変更されることがあります。その結果、希望する状態と報告される状態が、ローカルシャドウサービスと AWS IoT クラウドで異なります

シャドウマネージャーがシャドウを同期すると、次の動作に従って変更がマージされます。

- v2.2.0 より前のバージョンのシャドウマネージャーを使用している場合、またはシャドウ同期方向を betweenDeviceAndCloud に指定している場合、次の動作が適用されます。
  - シャドウの希望する状態でマージの競合が発生すると、シャドウマネージャーはローカルシャドウドキュメントの競合するセクションを AWS IoT クラウドからの値で上書きします。
  - シャドウの報告された状態でマージ競合が発生すると、シャドウマネージャーは、AWS IoT クラウドのシャドウの競合するセクションを、ローカルシャドウドキュメントからの値で上書きします。

- シャドウ同期方向に `deviceToCloud` を指定すると、シャドウマネージャーは、AWS IoT クラウド内のシャドウの競合するセクションをローカルシャドウドキュメントからの値で上書きします。
- シャドウ同期方向に `cloudToDevice` を指定すると、シャドウマネージャーは、ローカルシャドウドキュメントの競合するセクションを AWS IoT クラウドからの値で上書きします。

Greengrass コアデバイスでのデータストリームの管理

AWS IoT Greengrass ストリームマネージャーは、大量の IoT データを AWS クラウド に転送する効率と信頼性を向上させます。ストリームマネージャーは、AWS クラウド にデータストリームをエクスポートする前に、AWS IoT Greengrass コアで処理します。ストリームマネージャーは、機械学習 (ML) の推論など、一般的なエッジシナリオと統合し、AWS IoT Greengrass コアデバイスがデータを AWS クラウド またはローカルストレージの送信先にエクスポートする前に、データを処理して分析します。

ストリームマネージャーは共通のインターフェースを提供して、カスタムコンポーネント開発を簡素化するため、カスタムストリーム管理機能を構築する必要はありません。コンポーネントは、標準化されたメカニズムを使用して大量のストリームを処理して、ローカルデータ保持ポリシーを管理できます。ストレージタイプ、サイズ、データ保持に関するポリシーをストリームごとに定義して、ストリームマネージャーがデータを処理とエクスポートする方法を制御できます。

ストリームマネージャーは、断続的または制限された接続環境で動作します。帯域幅の使用、タイムアウト動作、AWS IoT Greengrass Core が接続または切断されたときのストリームデータの処理方法を定義できます。優先順位を設定して、AWS IoT Greengrass コアがストリームを AWS クラウドにエクスポートする順位も制御できます。これにより、重要なデータを他のデータよりも早く処理できるようになります。

保存またはさらなる処理と分析するため、ストリームマネージャーが AWS クラウド にデータを自動的にエクスポートするように設定できます。ストリームマネージャーは、次の AWS クラウド 送信先へのエクスポートをサポートしています:

- AWS IoT Analytics のチャネル。AWS IoT Analytics はデータに高度な分析を可能にし、ビジネス上の判断と機械学習モデルの改善に役立ちます。詳細については、『AWS IoT Analytics ユーザーガイド』の「What is AWS IoT Analytics? (とは ?)」 を参照してください。
- Amazon Kinesis Data Streams のストリーム。Kinesis Data Streams を使用して大量のデータを集約し、データウェアハウスまたは MapReduce クラスターにロードできます。詳細については、Amazon Kinesis Data Streams デベロッパーガイドの「Amazon Kinesis Data Streams とは」を参照してください。
- AWS IoT SiteWise のアセットプロパティ。AWS IoT SiteWise は、産業機器のデータを大規模に収集、整理、分析することを可能にします。詳細については、『AWS IoT SiteWise ユーザーガイド』の「What is AWS IoT SiteWise? (とは ?)」 を参照してください。

- Amazon Simple Storage Service (Amazon S3) のオブジェクト。Amazon S3 を使用して大量のデータの保存と取得を行えます。詳細については、「Amazon Simple Storage Service デベロッパーガイド」の「Amazon S3 とは」参照してください。

ストリーム管理ワークフロー

IoT アプリケーションは、ストリームマネージャー SDK を介してストリームマネージャーとやり取りします。

単純なワークフローでは、AWS IoT Greengrass コアのコンポーネントは、時系列温度と圧力メトリクスなど、IoT データを消費します。コンポーネントは、データをフィルタリングまたは圧縮し、ストリームマネージャー SDK を呼び出して、ストリームマネージャーのストリームにデータを書き込むことがあります。ストリームマネージャーは、ストリームに定義したポリシーに基づいて、ストリームを自動的に AWS クラウド にエクスポートできます。コンポーネントは、ローカルデータベースまたはストレージリポジトリにデータを直接送信することもできます。

IoT アプリケーションは、ストリームに読み書きを行う複数のカスタムコンポーネントを含めることができます。これらのコンポーネントは、ストリームに読み書きを行って、AWS IoT Greengrass コアデバイスにデータのフィルタリング、集約、分析できます。これにより、コアから AWS クラウドまたはローカルの送信先にデータを転送する前に、ローカルイベントに迅速に対応して貴重な情報を抽出することが可能になります。

開始するには、ストリームマネージャーコンポーネントを AWS IoT Greengrass コアデバイスに展開してください。デプロイで、ストリームマネージャーコンポーネントのパラメータを設定して、Greengrass コアデバイスのすべてのストリームに適用される設定を定義します。これらのパラメータを使用し、ビジネスニーズと環境の制約に基づいて、ストリームマネージャーがストリームを保存、処理、エクスポートする方法を制御します。

ストリームマネージャーを設定したら、IoT アプリケーションを作成してデプロイできます。これらは通常、ストリームを作成してやり取りするため、ストリームマネージャー SDK で `StreamManagerClient` を使用するカスタムコンポーネントです。ストリームを作成するとき、エクスポート送信先、優先度、永続性など、ストリームごとのポリシーを定義できます。

要件

ストリームマネージャーを使用する際に次の要件が適用されます。

- ストリームマネージャーは、AWS IoT Greengrass Core ソフトウェアに加えて、最低 70 MB の RAM が必要です。合計メモリ要件は、ワークロードによって異なります。
- AWS IoT Greengrass コンポーネントは、ストリームマネージャーとやり取りするためにストリームマネージャー SDK を使用する必要があります。ストリームマネージャー SDK は次の言語で利用可能です。
  - Java 用ストリームマネージャー SDK (v1.1.0 以降)
  - Node.js 用ストリームマネージャー SDK (v1.1.0 以降)
  - Python 用ストリームマネージャー SDK (v1.1.0 以降)
- AWS IoT Greengrass コンポーネントはストリームマネージャーを使用するため、ストリームマネージャーコンポーネント (`aws.greengrass.StreamManager`) をレシピで従属関係として指定する必要があります。

> **Note**
>
> ストリームマネージャーを使用してデータをクラウドにエクスポートする場合、ストリームマネージャーコンポーネントのバージョン 2.0.7 を v2.0.8 と v2.0.11 の間のバージョンにアップグレードすることはできません。ストリームマネージャーを初めてデプロイする場合、ストリームマネージャーコンポーネントの最新バージョンをデプロイすることを強くお勧めします。

- ストリームに対して AWS クラウド のエクスポート先を定義する場合、エクスポートターゲットを作成して、Greengrass デバイスロールでアクセス許可を付与する必要があります。送信先により、他の要件も適用される場合があります。詳細については、以下を参照してください。
  - the section called “AWS IoT Analytics チャネル”
  - the section called “Amazon Kinesis Data Streams”
  - the section called “AWS IoT SiteWise アセットプロパティ”
  - the section called “Amazon S3 オブジェクト”

  これらの AWS クラウド リソースを維持する責任があります。

データセキュリティ

ストリームマネージャーを使用する場合は、次のセキュリティ上の考慮事項に注意してください。

ローカルデータセキュリティ

AWS IoT Greengrass は、コアデバイスのローカルコンポーネント間で保管中または転送中のストリームデータを暗号化しません。

- 保管時のデータ。ストリームデータは、ストレージディレクトリにローカルに保存されます。データセキュリティのため、AWS IoT Greengrass は ファイル許可とフルディスク暗号化が有効になっている場合に依存します。オプションの STREAM_MANAGER_STORE_ROOT_DIR パラメータを使用して、ストレージディレクトリを指定できます。後でこのパラメータを変更して別のストレージディレクトリを使用した場合、AWS IoT Greengrass は以前のストレージディレクトリまたはその内容を削除しません。
- ローカルで転送中のデータ。AWS IoT Greengrass は、データソース、AWS IoT Greengrass コンポーネント、ストリームマネージャー間でローカル転送中のストリームデータを暗号化しません。
- AWS クラウド への転送中のデータ。ストリームマネージャーによって AWS クラウド にエクスポートされたデータストリームは、Transport Layer Security (TLS) を備えた標準 AWS サービスクライアント暗号化を使用します。

クライアント承認

ストリーム マネージャー クライアントは、ストリームマネージャー SDK を使用してストリームマネージャーと通信します。クライアント認証が有効になっているとき、Greengrass コンポーネントのみがストリームマネージャーのストリームとやり取りできます。クライアント認証が無効になっている場合、Greengrass コアデバイスで実行されているすべてのプロセスは、ストリームマネージャーのストリームとやり取りできます。ビジネスケースで要求される場合にのみ、認証を無効にする必要があります。

クライアント認証モードを設定するには、STREAM_MANAGER_AUTHENTICATE_CLIENT パラメータを使用します。ストリームマネージャーコンポーネントをコアデバイスにデプロイするとき、このパラメータを設定できます。

	有効	無効
パラメータ値	`true` (デフォルトおよび推奨)	`false`
許可されるクライアント	コアデバイスの Greengrass コンポーネント	コアデバイスの Greengrass コンポーネント

	有効	無効
		Greengrass コアデバイスで実行されているその他のプロセス

以下も参照してください。

- the section called “ストリームマネージャーの設定”
- the section called “ StreamManagerClient を使用してストリームを操作する”
- the section called “クラウドでサポートされている送信先のエクスポート設定”

ストリームマネージャーを使用するカスタムコンポーネントを作成する

IoT デバイスデータを保存、処理、エクスポートするため、カスタム Greengrass コンポーネントのストリームマネージャーを使用します。このセクションの手順と例を使用して、ストリームマネージャーと連携するコンポーネントレシピ、アーティファクト、アプリケーションを作成します。コンポーネントを開発してテストする方法の詳細については、「AWS IoT Greengrass コンポーネントの作成」を参照してください。

トピック

- ストリームマネージャーを使用するコンポーネントレシピの定義
- アプリケーションコードでストリームマネージャーに接続

ストリームマネージャーを使用するコンポーネントレシピの定義

カスタムコンポーネントでストリームマネージャーを使用するには、`aws.greengrass.StreamManager` コンポーネントを従属関係として定義する必要があります。ストリームマネージャー SDK も提供する必要があります。ストリームマネージャー SDK を任意の言語でダウンロードして使用するには、次のタスクを実行してください。

Java 用ストリームマネージャー SDK の使用

Java 用ストリームマネージャー SDK は、コンポーネントのコンパイルに利用可能な JAR ファイルとして利用できます。次に、ストリームマネージャー SDK を含むアプリケーション JAR を作成し、アプリケーション JAR をコンポーネントアーティファクトとして定義して、コンポーネントのライフサイクルでアプリケーション JAR を実行できます。

Java 用ストリームマネージャー SDK を使用するには

1. Java JAR ファイル用ストリームマネージャー SDK をダウンロードします。
2. Java アプリケーションと ストリームマネージャー SDK JAR ファイルからコンポーネントアーティファクトを作成するため、次のいずれかを実行します:
   - ストリームマネージャー SDK JAR を含む JAR ファイルとしてアプリケーションを構築し、この JAR ファイルをコンポーネントレシピで実行します。
   - ストリームマネージャー SDK JAR をコンポーネントアーティファクトとして定義します。コンポーネントレシピでアプリケーションを実行するとき、そのアーティファクトをクラスパスに追加します。

   コンポーネントレシピは、次の例のようになります。このコンポーネントは、修正された StreamManagerS3.java のバージョンの例を実行し、`StreamManagerS3.jar` にストリームマネージャー SDK JAR が含まれています。

   JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.StreamManagerS3Java",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Uses stream manager to upload a file to an S3
 bucket.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.StreamManager": {
      "VersionRequirement": "^2.0.0"
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "run": "java -jar {artifacts:path}/StreamManagerS3.jar"
```

```
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Java/1.0.0/StreamManagerS3.jar"
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.StreamManagerS3Java
ComponentVersion: 1.0.0
ComponentDescription: Uses stream manager to upload a file to an S3 bucket.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.StreamManager:
    VersionRequirement: "^2.0.0"
Manifests:
  - Lifecycle:
      run: java -jar {artifacts:path}/StreamManagerS3.jar
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Java/1.0.0/StreamManagerS3.jar
```

コンポーネントを開発してテストする方法の詳細については、「AWS IoT Greengrass コンポーネントの作成」を参照してください。

Python 用ストリームマネージャー SDK を使用

Python 用ストリームマネージャー SDK は、コンポーネントに含めることが可能なソースコードとして利用できます。ストリームマネージャー SDK の ZIP ファイルを作成し、コンポーネントアーティファクトとして ZIP ファイルを定義して、コンポーネントのライフサイクルに SDK の要件をインストールします。

Python 用ストリームマネージャー SDK を使用するには

1. aws-greengrass–stream-manager-sdk-python リポジトリをクローンまたはダウンロードします。

```
git clone git@github.com:aws-greengrass/aws-greengrass-stream-manager-sdk-python.git
```

2. Python 用ストリームマネージャー SDK のソースコードが含まれる `stream_manager` フォルダを含む ZIP ファイルを作成します。AWS IoT Greengrass Core ソフトウェアがコンポーネントをインストールする際に解凍するコンポーネントアーティファクトとして、この ZIP ファイルを提供できます。次のコマンドを実行します

   a. 前のステップでクローンまたはダウンロードしたリポジトリを含むフォルダを開きます。

   ```
   cd aws-greengrass-stream-manager-sdk-python
   ```

   b. `stream_manager_sdk.zip` という名前の ZIP ファイルに `stream_manager` フォルダを圧縮します。

   Linux or Unix

   ```
   zip -rv stream_manager_sdk.zip stream_manager
   ```

   Windows Command Prompt (CMD)

   ```
   tar -acvf stream_manager_sdk.zip stream_manager
   ```

   PowerShell

   ```
   Compress-Archive stream_manager stream_manager_sdk.zip
   ```

   c. `stream_manager_sdk.zip` ファイルに `stream_manager` フォルダとそのコンテンツが含まれていることを確認します。次のコマンドを実行して、ZIP ファイルのコンテンツを一覧表示します。

   Linux or Unix

   ```
   unzip -l stream_manager_sdk.zip
   ```

Windows Command Prompt (CMD)

```
tar -tf stream_manager_sdk.zip
```

出力は以下の例のようになります。

```
Archive:  aws-greengrass-stream-manager-sdk-python/stream_manager.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  02-24-2021 20:45   stream_manager/
      913  02-24-2021 20:45   stream_manager/__init__.py
     9719  02-24-2021 20:45   stream_manager/utilinternal.py
     1412  02-24-2021 20:45   stream_manager/exceptions.py
     1004  02-24-2021 20:45   stream_manager/util.py
        0  02-24-2021 20:45   stream_manager/data/
   254463  02-24-2021 20:45   stream_manager/data/__init__.py
    26515  02-24-2021 20:45   stream_manager/streammanagerclient.py
---------                     -------
   294026                     8 files
```

3. ストリームマネージャー SDK アーティファクトをコンポーネントのアーティファクトフォルダにコピーします。ストリームマネージャー SDK ZIP ファイルに加えて、コンポーネントは SDK の `requirements.txt` ファイルを使用して、ストリームマネージャー SDK の従属関係をインストールします。*`~/greengrass-components`* をローカル開発に使用するフォルダへのパスに置き換えます。

   Linux or Unix

   ```
   cp {stream_manager_sdk.zip,requirements.txt} ~/greengrass-components/artifacts/
   com.example.StreamManagerS3Python/1.0.0/
   ```

   Windows Command Prompt (CMD)

   ```
   robocopy . %USERPROFILE%\greengrass-components\artifacts
   \com.example.StreamManagerS3Python\1.0.0 stream_manager_sdk.zip
   robocopy . %USERPROFILE%\greengrass-components\artifacts
   \com.example.StreamManagerS3Python\1.0.0 requirements.txt
   ```

PowerShell

```
cp .\stream_manager_sdk.zip,.\requirements.txt ~\greengrass-components\artifacts
\com.example.StreamManagerS3Python\1.0.0\
```

4. コンポーネントレシピを作成します。レシピで次の手順を実行します:

   a. `stream_manager_sdk.zip` と `requirements.txt` をアーティファクトとして定義します。

   b. Python アプリケーションをアーティファクトとして定義します。

   c. インストールライフサイクルで、ストリームマネージャー SDK の要件を `requirements.txt` からインストールします。

   d. 実行ライフサイクルで、ストリームマネージャー SDK を PYTHONPATH に追加して、Python アプリケーションを実行します。

   コンポーネントレシピは、次の例のようになります。このコンポーネントは stream_manager_s3.py の例を実行します。

   JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.StreamManagerS3Python",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Uses stream manager to upload a file to an S3
 bucket.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.StreamManager": {
      "VersionRequirement": "^2.0.0"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "pip3 install --user -r {artifacts:path}/requirements.txt",
```

```
        "run": "export PYTHONPATH=$PYTHONPATH:{artifacts:decompressedPath}/
stream_manager_sdk; python3 {artifacts:path}/stream_manager_s3.py"
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_sdk.zip",
          "Unarchive": "ZIP"
        },
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_s3.py"
        },
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Python/1.0.0/requirements.txt"
        }
      ]
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "pip3 install --user -r {artifacts:path}/requirements.txt",
        "run": "set \"PYTHONPATH=%PYTHONPATH%;{artifacts:decompressedPath}/
stream_manager_sdk\" & py -3 {artifacts:path}/stream_manager_s3.py"
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_sdk.zip",
          "Unarchive": "ZIP"
        },
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_s3.py"
        },
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Python/1.0.0/requirements.txt"
        }
      ]
    }
```

```
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.StreamManagerS3Python
ComponentVersion: 1.0.0
ComponentDescription: Uses stream manager to upload a file to an S3 bucket.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.StreamManager:
    VersionRequirement: "^2.0.0"
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: pip3 install --user -r {artifacts:path}/requirements.txt
      run: |
        export PYTHONPATH=$PYTHONPATH:{artifacts:decompressedPath}/
stream_manager_sdk
        python3 {artifacts:path}/stream_manager_s3.py
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_sdk.zip
        Unarchive: ZIP
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_s3.py
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Python/1.0.0/requirements.txt
  - Platform:
      os: windows
    Lifecycle:
      install: pip3 install --user -r {artifacts:path}/requirements.txt
      run: |
        set "PYTHONPATH=%PYTHONPATH%;{artifacts:decompressedPath}/
stream_manager_sdk"
        py -3 {artifacts:path}/stream_manager_s3.py
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_sdk.zip
        Unarchive: ZIP
```

```
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_s3.py
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3Python/1.0.0/requirements.txt
```

コンポーネントを開発してテストする方法の詳細については、「AWS IoT Greengrass コンポーネントの作成」を参照してください。

JavaScript 用ストリームマネージャー SDK の使用

JavaScript 用ストリームマネージャー SDK は、コンポーネントに含めることが可能なソースコードとして利用できます。ストリームマネージャー SDK の ZIP ファイルを作成し、コンポーネントアーティファクトとして ZIP ファイルを定義して、コンポーネントのライフサイクルに SDK をインストールします。

JavaScript 用ストリームマネージャー SDK を使用するには

1. aws-greengrass–stream-manager-sdk-js リポジトリをクローンまたはダウンロードします。

   ```
   git clone git@github.com:aws-greengrass/aws-greengrass-stream-manager-sdk-js.git
   ```

2. JavaScript 用ストリームマネージャー SDK のソースコードが含まれる `aws-greengrass-stream-manager-sdk` フォルダを含む ZIP ファイルを作成します。AWS IoT Greengrass Core ソフトウェアがコンポーネントをインストールする際に解凍するコンポーネントアーティファクトとして、この ZIP ファイルを提供できます。次のコマンドを実行します

   a. 前のステップでクローンまたはダウンロードしたリポジトリを含むフォルダを開きます。

      ```
      cd aws-greengrass-stream-manager-sdk-js
      ```

   b. `stream-manager-sdk.zip` という名前の ZIP ファイルに `aws-greengrass-stream-manager-sdk` フォルダを圧縮します。

      Linux or Unix

      ```
      zip -rv stream-manager-sdk.zip aws-greengrass-stream-manager-sdk
      ```

Windows Command Prompt (CMD)

```
tar -acvf stream-manager-sdk.zip aws-greengrass-stream-manager-sdk
```

PowerShell

```
Compress-Archive aws-greengrass-stream-manager-sdk stream-manager-sdk.zip
```

c. stream-manager-sdk.zip ファイルに aws-greengrass-stream-manager-sdk フォルダとそのコンテンツが含まれていることを確認します。次のコマンドを実行して、ZIP ファイルのコンテンツを一覧表示します。

Linux or Unix

```
unzip -l stream-manager-sdk.zip
```

Windows Command Prompt (CMD)

```
tar -tf stream-manager-sdk.zip
```

出力は以下の例のようになります。

```
Archive:  stream-manager-sdk.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/
      369  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/package.json
     1017  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/util.js
     8374  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/utilInternal.js
     1937  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/exceptions.js
        0  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/data/
   353343  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/data/index.js
    22599  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/client.js
      216  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/index.js
---------                     -------
   387855                     9 files
```

3. ストリームマネージャー SDK アーティファクトをコンポーネントのアーティファクトフォルダにコピーします。*~/greengrass-components* をローカル開発に使用するフォルダへのパスに置き換えます。

   Linux or Unix

   ```
   cp stream-manager-sdk.zip ~/greengrass-components/artifacts/
   com.example.StreamManagerS3JS/1.0.0/
   ```

   Windows Command Prompt (CMD)

   ```
   robocopy . %USERPROFILE%\greengrass-components\artifacts
   \com.example.StreamManagerS3JS\1.0.0 stream-manager-sdk.zip
   ```

   PowerShell

   ```
   cp .\stream-manager-sdk.zip ~\greengrass-components\artifacts
   \com.example.StreamManagerS3JS\1.0.0\
   ```

4. コンポーネントレシピを作成します。レシピで次の手順を実行します:

   a. `stream-manager-sdk.zip` をアーティファクトとして定義します。

   b. JavaScript アプリケーションをアーティファクトとして定義します。

   c. インストールライフサイクルで、`stream-manager-sdk.zip` アーティファクトからストリームマネージャー SDK をインストールします。この `npm install` コマンドは、ストリームマネージャー SDK とその従属関係を含む `node_modules` フォルダを作成します。

   d. 実行ライフサイクルで、`node_modules` フォルダを `NODE_PATH` に追加して、JavaScript アプリケーションを実行します。

   コンポーネントレシピは、次の例のようになります。このコンポーネントは、StreamManagerS3 の例を実行します。

   JSON

   ```
   {
     "RecipeFormatVersion": "2020-01-25",
     "ComponentName": "com.example.StreamManagerS3JS",
     "ComponentVersion": "1.0.0",
   ```

```
  "ComponentDescription": "Uses stream manager to upload a file to an S3
 bucket.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.StreamManager": {
      "VersionRequirement": "^2.0.0"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "npm install {artifacts:decompressedPath}/stream-manager-sdk/
aws-greengrass-stream-manager-sdk",
        "run": "export NODE_PATH=$NODE_PATH:{work:path}/node_modules; node
 {artifacts:path}/index.js"
      },
      "Artifacts": [
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3JS/1.0.0/stream-manager-sdk.zip",
          "Unarchive": "ZIP"
        },
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3JS/1.0.0/index.js"
        }
      ]
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "npm install {artifacts:decompressedPath}/stream-manager-sdk/
aws-greengrass-stream-manager-sdk",
        "run": "set \"NODE_PATH=%NODE_PATH%;{work:path}/node_modules\" & node
 {artifacts:path}/index.js"
      },
      "Artifacts": [
        {
```

```
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3JS/1.0.0/stream-manager-sdk.zip",
          "Unarchive": "ZIP"
        },
        {
          "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3JS/1.0.0/index.js"
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.StreamManagerS3JS
ComponentVersion: 1.0.0
ComponentDescription: Uses stream manager to upload a file to an S3 bucket.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.StreamManager:
    VersionRequirement: "^2.0.0"
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: npm install {artifacts:decompressedPath}/stream-manager-sdk/aws-
greengrass-stream-manager-sdk
      run: |
        export NODE_PATH=$NODE_PATH:{work:path}/node_modules
        node {artifacts:path}/index.js
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3JS/1.0.0/stream-manager-sdk.zip
        Unarchive: ZIP
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3JS/1.0.0/index.js
  - Platform:
      os: windows
    Lifecycle:
```

```
      install: npm install {artifacts:decompressedPath}/stream-manager-sdk/aws-
greengrass-stream-manager-sdk
      run: |
        set "NODE_PATH=%NODE_PATH%;{work:path}/node_modules"
        node {artifacts:path}/index.js
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3JS/1.0.0/stream-manager-sdk.zip
        Unarchive: ZIP
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3JS/1.0.0/index.js
```

コンポーネントを開発してテストする方法の詳細については、「AWS IoT Greengrass コンポーネントの作成」を参照してください。

アプリケーションコードでストリームマネージャーに接続

アプリケーションでストリームマネージャーに接続するには、ストリームマネージャー SDK から `StreamManagerClient` のインスタンスを作成します。このクライアントは、デフォルトのポート 8088 または指定したポートでストリーム マネージャー コンポーネントに接続します。インスタンスを作成した後に `StreamManagerClient` を使用する方法については、「 StreamManagerClient を使用してストリームを操作する」を参照してください。

Example 例: デフォルトポートでストリームマネージャーに接続

Java

```
import com.amazonaws.greengrass.streammanager.client.StreamManagerClient;

public class MyStreamManagerComponent {

    void connectToStreamManagerWithDefaultPort() {
        StreamManagerClient client = StreamManagerClientFactory.standard().build();

        // Use the client.
    }
}
```

Python

```
from stream_manager import (
    StreamManagerClient
)

def connect_to_stream_manager_with_default_port():
    client = StreamManagerClient()

    # Use the client.
```

JavaScript

```
const {
    StreamManagerClient
} = require('aws-greengrass-stream-manager-sdk');

function connectToStreamManagerWithDefaultPort() {
    const client = new StreamManagerClient();

    // Use the client.
}
```

Example 例: デフォルト以外のポートでストリームマネージャーに接続

ストリームマネージャーにデフォルト以外のポートを設定する場合、プロセス間通信を使用してコンポーネント設定からポートを取得する必要があります。

> **Note**
>
> port 設定パラメータは、ストリームマネージャーをデプロイする際に STREAM_MANAGER_SERVER_PORT で指定した値が含まれます。

Java

```
void connectToStreamManagerWithCustomPort() {
    EventStreamRPCConnection eventStreamRpcConnection =
 IPCUtils.getEventStreamRpcConnection();
    GreengrassCoreIPCClient greengrassCoreIPCClient = new
 GreengrassCoreIPCClient(eventStreamRpcConnection);
```

```
    List<String> keyPath = new ArrayList<>();
    keyPath.add("port");

    GetConfigurationRequest request = new GetConfigurationRequest();
    request.setComponentName("aws.greengrass.StreamManager");
    request.setKeyPath(keyPath);
    GetConfigurationResponse response =
            greengrassCoreIPCClient.getConfiguration(request,
 Optional.empty()).getResponse().get();
    String port = response.getValue().get("port").toString();
    System.out.print("Stream Manager is running on port: " + port);

    final StreamManagerClientConfig config = StreamManagerClientConfig.builder()

 .serverInfo(StreamManagerServerInfo.builder().port(Integer.parseInt(port)).build()).build()

    StreamManagerClient client =
 StreamManagerClientFactory.standard().withClientConfig(config).build();

    // Use the client.
}
```

Python

```
import awsiot.greengrasscoreipc
from awsiot.greengrasscoreipc.model import (
    GetConfigurationRequest
)
from stream_manager import (
    StreamManagerClient
)

TIMEOUT = 10

def connect_to_stream_manager_with_custom_port():
    # Use IPC to get the port from the stream manager component configuration.
    ipc_client = awsiot.greengrasscoreipc.connect()
    request = GetConfigurationRequest()
    request.component_name = "aws.greengrass.StreamManager"
    request.key_path = ["port"]
    operation = ipc_client.new_get_configuration()
    operation.activate(request)
    future_response = operation.get_response()
```

```
    response = future_response.result(TIMEOUT)
    stream_manager_port = str(response.value["port"])

    # Use port to create a stream manager client.
    stream_client = StreamManagerClient(port=stream_manager_port)

    # Use the client.
```

StreamManagerClient を使用してストリームを操作する

Greengrass コアデバイスで実行されるユーザー定義の Greengrass コンポーネントは、ストリームマネージャー SDK の `StreamManagerClient` オブジェクトを使用し、ストリームマネージャーでストリームを作成してストリームとやり取りします。コンポーネントがストリームを作成するとき、ストリームに対して AWS クラウド 送信先、優先順位、他のエクスポートとデータ保持ポリシーを定義します。ストリームマネージャーにデータを送信するため、コンポーネントはデータをストリームに追加します。ストリームにエクスポート先が定義されている場合、ストリームマネージャーは自動的にストリームをエクスポートします。

> **Note**
>
> 通常、ストリームマネージャーのクライアントはユーザー定義の Greengrass コンポーネントです。ビジネスケースで必要な場合、Greengrass コア (例えば、Docker コンテナなど) で実行されている非コンポーネントプロセスがストリームマネージャーとやり取りを可能にすることもできます。詳細については、「the section called “クライアント承認”」を参照してください。

このトピックのスニペットは、クライアントが `StreamManagerClient` 手法を使用してストリームを操作する方法を示しています。手法とその引数の実装に関する詳細については、各スニペットの下に記載されている SDK リファレンスへのリンクを使用してください。

Lambda 関数でストリームマネージャーを使用する場合、Lambda 関数は `StreamManagerClient` を関数ハンドらの外側でインスタンス化する必要があります。ハンドラでインスタンス化されると、関数は呼び出されるたびに `client` およびストリームマネージャへの接続を作成します。

> **Note**
>
> ハンドラで StreamManagerClient のインスタンス化を行う場合は、client が作業を完了したときに、close() メソッドを明示的に呼び出す必要があります。それ以外の場合、client は接続を開いたままにし、スクリプトが終了するまで別のスレッドを実行します。

StreamManagerClient では次の操作がサポートされています。

- the section called “メッセージストリームの作成”
- the section called “メッセージの追加”
- the section called “メッセージの読み取り”
- the section called “ストリームの一覧表示”
- the section called “メッセージストリームの説明”
- the section called “メッセージストリームの更新”
- the section called “メッセージストリームの削除”

メッセージストリームの作成

ストリームを作成するには、ユーザー定義の Greengasss コンポーネントが作成手法を呼び出して MessageStreamDefinition オブジェクトに渡します。このオブジェクトは、ストリームの一意の名前を指定し、最大ストリームサイズに達したときにストリームマネージャーが新しいデータを処理する方法を定義します。MessageStreamDefinition とそのデータ型 (ExportDefinition、StrategyOnFull、Persistence など) を使用して、他のストリームプロパティを定義できます。具体的には次のとおりです。

- ターゲット AWS IoT Analytics、Kinesis データストリーム、AWS IoT SiteWise、自動エクスポート用の Amazon S3 送信先。詳細については、「the section called “クラウドでサポートされている送信先のエクスポート設定”」を参照してください。
- エクスポートの優先度。ストリームマネージャーは、プライオリティの低いストリームよりも先にプライオリティの高いストリームをエクスポートします。
- AWS IoT Analytics の最大バッチサイズとバッチ間隔、Kinesis データストリーム、AWS IoT SiteWise 送信先。ストリームマネージャーは、いずれかの条件が満たされたときにメッセージをエクスポートします。

- T ime-to-live (TTL)。ストリームデータが処理可能であることを保証する時間。この期間内にデータを消費できることを確認する必要があります。これは削除ポリシーではありません。TTL 期間の直後にデータが削除されない場合があります。
- ストリームの永続性。ストリームをファイルシステムに保存して、コアを再起動してもデータを保持するか、ストリームをメモリに保存するかを選択します。
- 開始するシーケンス番号。エクスポートの開始メッセージとして使用するメッセージのシーケンス番号を指定します。

`MessageStreamDefinition` の詳細については、対象言語の SDK リファレンスを参照してください。

- MessageStreamDefinition Java SDK の
- MessageStreamDefinition Node.js SDK の
- MessageStreamDefinition Python SDK の

> **Note**
>
> `StreamManagerClient` は、ストリームを HTTP サーバーにエクスポートするため、ターゲットの送信先も提供します。このターゲットは、テストのみを目的としています。本場環境での使用は安定しておらず、サポートされていません。

ストリームが作成されると、Greengrass コンポーネントはストリームにメッセージを追加し、エクスポート用データを送信して、ローカル処理用にストリームからメッセージを読み取ります。作成するストリームの数は、ハードウェアの機能とビジネスケースによって異なります。1 つの戦略は、AWS IoT Analytics または Kinesis データストリームのターゲットチャネルごとにストリームを作成することです。ただし、1 つのストリームに複数のターゲットを定義できます。ストリームは寿命に耐久性があります。

要件

この操作には次の要件があります:

- ストリームマネージャー SDK の最小バージョン: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

例

次のスニペットでは、StreamName という名前のストリームが作成されます。MessageStreamDefinition のストリームプロパティとと下位のデータタイプを定義します。

Python

```
client = StreamManagerClient()

try:
    client.create_message_stream(MessageStreamDefinition(
        name="StreamName",    # Required.
        max_size=268435456,    # Default is 256 MB.
        stream_segment_size=16777216,    # Default is 16 MB.
        time_to_live_millis=None,    # By default, no TTL is enabled.
        strategy_on_full=StrategyOnFull.OverwriteOldestData,    # Required.
        persistence=Persistence.File,    # Default is File.
        flush_on_write=False,    # Default is false.
        export_definition=ExportDefinition(    # Optional. Choose where/how the
 stream is exported to the AWS ####.
            kinesis=None,
            iot_analytics=None,
            iot_sitewise=None,
            s3_task_executor=None
        )
    ))
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: create_message_stream | MessageStreamDefinition

Java

```
try (final StreamManagerClient client =
 StreamManagerClientFactory.standard().build()) {
    client.createMessageStream(
            new MessageStreamDefinition()
                    .withName("StreamName") // Required.
```

```
                    .withMaxSize(268435456L)    // Default is 256 MB.
                    .withStreamSegmentSize(16777216L)    // Default is 16 MB.
                    .withTimeToLiveMillis(null)    // By default, no TTL is enabled.
                    .withStrategyOnFull(StrategyOnFull.OverwriteOldestData)    //
 Required.
                    .withPersistence(Persistence.File)    // Default is File.
                    .withFlushOnWrite(false)    // Default is false.
                    .withExportDefinition(    // Optional. Choose where/how the
 stream is exported to the AWS ####.
                            new ExportDefinition()
                                    .withKinesis(null)
                                    .withIotAnalytics(null)
                                    .withIotSitewise(null)
                                    .withS3(null)
                    )

    );
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: createMessageStream | MessageStreamDefinition

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
  try {
    await client.createMessageStream(
      new MessageStreamDefinition()
        .withName("StreamName") // Required.
        .withMaxSize(268435456)  // Default is 256 MB.
        .withStreamSegmentSize(16777216)  // Default is 16 MB.
        .withTimeToLiveMillis(null)  // By default, no TTL is enabled.
        .withStrategyOnFull(StrategyOnFull.OverwriteOldestData)  // Required.
        .withPersistence(Persistence.File)  // Default is File.
        .withFlushOnWrite(false)  // Default is false.
        .withExportDefinition(  // Optional. Choose where/how the stream is exported
 to the AWS ####.
          new ExportDefinition()
            .withKinesis(null)
            .withIotAnalytics(null)
            .withIotSiteWise(null)
            .withS3(null)
```

```
        )
    );
  } catch (e) {
    // Properly handle errors.
  }
});
client.onError((err) => {
  // Properly handle connection errors.
  // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: createMessageStream | MessageStreamDefinition

エクスポート先設定の詳細については、「the section called “クラウドでサポートされている送信先のエクスポート設定”」を参照してください。

メッセージの追加

エクスポート用にストリームマネージャーにデータを送信するため、Greengrass コンポーネントはデータをターゲットストリームに追加します。エクスポート先によって、この手法に渡すデータタイプが決まります。

要件

この操作には次の要件があります:

- ストリームマネージャー SDK の最小バージョン: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

例

AWS IoT Analytics または Kinesis データストリームのエクスポート先

次のスニペットは、`StreamName` という名前のストリームにメッセージを追加します。AWS IoT Analytics または Kinesis データストリストリームの送信先の場合、Greengrass コンポーネントがデータの BLOB を追加します。

このスニペットには次の要件があります:

- ストリームマネージャー SDK の最小バージョン: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Python

```
client = StreamManagerClient()

try:
    sequence_number = client.append_message(stream_name="StreamName",
 data=b'Arbitrary bytes data')
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: append_message

Java

```
try (final StreamManagerClient client =
 StreamManagerClientFactory.standard().build()) {
    long sequenceNumber = client.appendMessage("StreamName", "Arbitrary byte
 array".getBytes());
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: appendMessage

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const sequenceNumber = await client.appendMessage("StreamName",
 Buffer.from("Arbitrary byte array"));
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: appendMessage

AWS IoT SiteWise エクスポート先

次のスニペットは、StreamName という名前のストリームにメッセージを追加します。AWS IoT SiteWise 送信先の場合、Greengrass コンポーネントはシリアル化された PutAssetPropertyValueEntry オブジェクトを追加します。詳細については、「the section called “AWS IoT SiteWise にエクスポートする”」を参照してください。

> **Note**
>
> AWS IoT SiteWise にデータを送信するとき、データは BatchPutAssetPropertyValue アクションの要件を満す必要があります。詳細については、AWS IoT SiteWise API リファレンスの「BatchPutAssetPropertyValue」を参照してください。

このスニペットには次の要件があります:

- ストリームマネージャー SDK の最小バージョン: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Python

```
client = StreamManagerClient()

try:
    # SiteWise requires unique timestamps in all messages and also needs timestamps
 not earlier
    # than 10 minutes in the past. Add some randomness to time and offset.

    # Note: To create a new asset property data, you should use the classes defined
 in the
    # greengrasssdk.stream_manager module.

    time_in_nanos = TimeInNanos(
        time_in_seconds=calendar.timegm(time.gmtime()) - random.randint(0, 60),
 offset_in_nanos=random.randint(0, 10000)
    )
    variant = Variant(double_value=random.random())
    asset = [AssetPropertyValue(value=variant, quality=Quality.GOOD,
 timestamp=time_in_nanos)]
```

```
    putAssetPropertyValueEntry =
 PutAssetPropertyValueEntry(entry_id=str(uuid.uuid4()),
 property_alias="PropertyAlias", property_values=asset)
    sequence_number = client.append_message(stream_name="StreamName",
 Util.validate_and_serialize_to_json_bytes(putAssetPropertyValueEntry))
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: append_message | PutAssetPropertyValueEntry

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    Random rand = new Random();
    // Note: To create a new asset property data, you should use the classes defined
 in the
    // com.amazonaws.greengrass.streammanager.model.sitewise package.
    List<AssetPropertyValue> entries = new ArrayList<>() ;

    // IoTSiteWise requires unique timestamps in all messages and also needs
 timestamps not earlier
    // than 10 minutes in the past. Add some randomness to time and offset.
    final int maxTimeRandomness = 60;
    final int maxOffsetRandomness = 10000;
    double randomValue = rand.nextDouble();
    TimeInNanos timestamp = new TimeInNanos()
            .withTimeInSeconds(Instant.now().getEpochSecond() -
 rand.nextInt(maxTimeRandomness))
            .withOffsetInNanos((long) (rand.nextInt(maxOffsetRandomness)));
    AssetPropertyValue entry = new AssetPropertyValue()
            .withValue(new Variant().withDoubleValue(randomValue))
            .withQuality(Quality.GOOD)
            .withTimestamp(timestamp);
    entries.add(entry);

    PutAssetPropertyValueEntry putAssetPropertyValueEntry = new
 PutAssetPropertyValueEntry()
            .withEntryId(UUID.randomUUID().toString())
            .withPropertyAlias("PropertyAlias")
```

```
            .withPropertyValues(entries);
    long sequenceNumber = client.appendMessage("StreamName",
 ValidateAndSerialize.validateAndSerializeToJsonBytes(putAssetPropertyValueEntry));
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: appendMessage | PutAssetPropertyValueEntry

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const maxTimeRandomness = 60;
        const maxOffsetRandomness = 10000;
        const randomValue = Math.random();
        // Note: To create a new asset property data, you should use the classes
 defined in the
        // aws-greengrass-core-sdk StreamManager module.
        const timestamp = new TimeInNanos()
            .withTimeInSeconds(Math.round(Date.now() / 1000) -
 Math.floor(Math.random() - maxTimeRandomness))
            .withOffsetInNanos(Math.floor(Math.random() * maxOffsetRandomness));
        const entry = new AssetPropertyValue()
            .withValue(new Variant().withDoubleValue(randomValue))
            .withQuality(Quality.GOOD)
            .withTimestamp(timestamp);

        const putAssetPropertyValueEntry =    new PutAssetPropertyValueEntry()
            .withEntryId(`${ENTRY_ID_PREFIX}${i}`)
            .withPropertyAlias("PropertyAlias")
            .withPropertyValues([entry]);
        const sequenceNumber = await client.appendMessage("StreamName",
 util.validateAndSerializeToJsonBytes(putAssetPropertyValueEntry));
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: appendMessage | PutAssetPropertyValueEntry

Amazon S3 のエクスポート先

次のスニペットは、StreamName という名前のストリームにエクスポートタスクを追加します。Amazon S3 の送信先の場合、Greengrass コンポーネントはシリアル化された S3ExportTaskDefinition オブジェクトを追加します。これには、ソース入力ファイルとターゲット Amazon S3 オブジェクトに関する情報が含まれています。指定されたオブジェクトが存在しない場合、ストリームマネージャーがユーザーに代わってそのオブジェクトを作成します。詳細については、「the section called “Amazon S3 へのエクスポート”」を参照してください。

このスニペットには次の要件があります:

- ストリームマネージャー SDK の最小バージョン: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Python

```
client = StreamManagerClient()

try:
    # Append an Amazon S3 Task definition and print the sequence number.
    s3_export_task_definition = S3ExportTaskDefinition(input_url="URLToFile",
 bucket="BucketName", key="KeyName")
    sequence_number = client.append_message(stream_name="StreamName",
 Util.validate_and_serialize_to_json_bytes(s3_export_task_definition))
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: append_message | S3ExportTaskDefinition

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    // Append an Amazon S3 export task definition and print the sequence number.
    S3ExportTaskDefinition s3ExportTaskDefinition = new S3ExportTaskDefinition()
        .withBucket("BucketName")
```

```
        .withKey("KeyName")
        .withInputUrl("URLToFile");
    long sequenceNumber = client.appendMessage("StreamName",
 ValidateAndSerialize.validateAndSerializeToJsonBytes(s3ExportTaskDefinition));
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: appendMessage | S3ExportTaskDefinition

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
     // Append an Amazon S3 export task definition and print the sequence number.
     const taskDefinition = new S3ExportTaskDefinition()
        .withBucket("BucketName")
        .withKey("KeyName")
        .withInputUrl("URLToFile");
        const sequenceNumber = await client.appendMessage("StreamName",
 util.validateAndSerializeToJsonBytes(taskDefinition)));
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: appendMessage | S3ExportTaskDefinition

メッセージの読み取り

ストリームのメッセージを読み取ります。

要件

この操作には次の要件があります:

- ストリームマネージャー SDK の最小バージョン: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

例

次のスニペットは、StreamName という名前のストリームからメッセージを読み取ります。読み取りメソッドは、読み取りを開始するシーケンス番号、読み取る最小値と最大値、メッセージを読み取るためのタイムアウトを指定するオプションの ReadMessagesOptions オブジェクトを受け取ります。

Python

```
client = StreamManagerClient()

try:
    message_list = client.read_messages(
        stream_name="StreamName",
        # By default, if no options are specified, it tries to read one message from
 the beginning of the stream.
        options=ReadMessagesOptions(
            desired_start_sequence_number=100,
            # Try to read from sequence number 100 or greater. By default, this is
 0.
            min_message_count=10,
            # Try to read 10 messages. If 10 messages are not available, then
 NotEnoughMessagesException is raised. By default, this is 1.
            max_message_count=100,    # Accept up to 100 messages. By default this
 is 1.
            read_timeout_millis=5000
            # Try to wait at most 5 seconds for the min_messsage_count to be
 fulfilled. By default, this is 0, which immediately returns the messages or an
 exception.
        )
    )
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: read_messages | ReadMessagesOptions

Java

```
try (final StreamManagerClient client =
 StreamManagerClientFactory.standard().build()) {
    List<Message> messages = client.readMessages("StreamName",
            // By default, if no options are specified, it tries to read one message
 from the beginning of the stream.
            new ReadMessagesOptions()
                    // Try to read from sequence number 100 or greater. By default
 this is 0.
                    .withDesiredStartSequenceNumber(100L)
                    // Try to read 10 messages. If 10 messages are not available,
 then NotEnoughMessagesException is raised. By default, this is 1.
                    .withMinMessageCount(10L)
                    // Accept up to 100 messages. By default this is 1.
                    .withMaxMessageCount(100L)
                    // Try to wait at most 5 seconds for the min_messsage_count to
 be fulfilled. By default, this is 0, which immediately returns the messages or an
 exception.
                    .withReadTimeoutMillis(Duration.ofSeconds(5L).toMillis())
    );
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: readMessages | ReadMessagesOptions

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const messages = await client.readMessages("StreamName",
            // By default, if no options are specified, it tries to read one message
 from the beginning of the stream.
            new ReadMessagesOptions()
                // Try to read from sequence number 100 or greater. By default this
 is 0.
                .withDesiredStartSequenceNumber(100)
                // Try to read 10 messages. If 10 messages are not available, then
 NotEnoughMessagesException is thrown. By default, this is 1.
                .withMinMessageCount(10)
                // Accept up to 100 messages. By default this is 1.
                .withMaxMessageCount(100)
```

```
                // Try to wait at most 5 seconds for the minMessageCount to be
 fulfilled. By default, this is 0, which immediately returns the messages or an
 exception.
                .withReadTimeoutMillis(5 * 1000)
        );
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: readMessages | ReadMessagesOptions

ストリームの一覧表示

ストリームマネージャーでストリームのリストを取得します。

要件

この操作には次の要件があります:

- ストリームマネージャー SDK の最小バージョン: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

例

次のスニペットは、ストリームマネージャーのストリームのリストを (名前で) 取得します。

Python

```
client = StreamManagerClient()

try:
    stream_names = client.list_streams()
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
```

```
    # Properly handle errors.
```

Python SDK リファレンス: list_streams

Java

```
try (final StreamManagerClient client =
 StreamManagerClientFactory.standard().build()) {
    List<String> streamNames = client.listStreams();
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: listStreams

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const streams = await client.listStreams();
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: listStreams

メッセージストリームの説明

ストリーム定義、サイズ、エクスポート状態を含め、ストリームに関するメタデータを取得します。

要件

この操作には次の要件があります:

- ストリームマネージャー SDK の最小バージョン: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

例

次のスニペットは、ストリームの定義、サイズ、エクスポーターのステータスなど、StreamName という名前のストリームに関するメタデータを取得します。

Python

```
client = StreamManagerClient()

try:
    stream_description = client.describe_message_stream(stream_name="StreamName")
    if stream_description.export_statuses[0].error_message:
        # The last export of export destination 0 failed with some error
        # Here is the last sequence number that was successfully exported
        stream_description.export_statuses[0].last_exported_sequence_number

    if (stream_description.storage_status.newest_sequence_number >
            stream_description.export_statuses[0].last_exported_sequence_number):
        pass
        # The end of the stream is ahead of the last exported sequence number
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: describe_message_stream

Java

```
try (final StreamManagerClient client =
 StreamManagerClientFactory.standard().build()) {
    MessageStreamInfo description = client.describeMessageStream("StreamName");
    String lastErrorMessage =
 description.getExportStatuses().get(0).getErrorMessage();
    if (lastErrorMessage != null && !lastErrorMessage.equals("")) {
        // The last export of export destination 0 failed with some error.
        // Here is the last sequence number that was successfully exported.
        description.getExportStatuses().get(0).getLastExportedSequenceNumber();
    }

    if (description.getStorageStatus().getNewestSequenceNumber() >
```

```
            description.getExportStatuses().get(0).getLastExportedSequenceNumber())
 {
        // The end of the stream is ahead of the last exported sequence number.
    }
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: describeMessageStream

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const description = await client.describeMessageStream("StreamName");
        const lastErrorMessage = description.exportStatuses[0].errorMessage;
        if (lastErrorMessage) {
            // The last export of export destination 0 failed with some error.
            // Here is the last sequence number that was successfully exported.
            description.exportStatuses[0].lastExportedSequenceNumber;
        }

        if (description.storageStatus.newestSequenceNumber >
            description.exportStatuses[0].lastExportedSequenceNumber) {
            // The end of the stream is ahead of the last exported sequence number.
        }
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: describeMessageStream

メッセージストリームの更新

既存ストリームのプロパティを更新します。ストリームを作成した後に要件が変更された場合、ストリームの更新が必要となる場合があります。例:

- AWS クラウド 送信先の新しいエクスポート設定を追加します。
- ストリームの最大サイズを増やして、データのエクスポートまたは保持方法を変更します。例えば、ストリームサイズとフル設定の戦略を組み合わせると、ストリームマネージャーが処理する前に、データが削除または拒否されることがあります。
- エクスポートの一時停止と再開。例えば、エクスポートタスクが長時間実行されているときにアップロードするデータ量を抑える場合。

Greengrass コンポーネントはこの高レベルプロセスに準拠してストリームを更新します:

1. ストリームの概要を取得します。
2. 対応する `MessageStreamDefinition` と下位オブジェクトのターゲットプロパティを更新します。
3. 更新済み `MessageStreamDefinition` として渡します。更新済みストリームの完全なオブジェクト定義を必ず含めてください。未定義のプロパティはデフォルト値に戻ります。

   エクスポートで開始メッセージとして使用するメッセージのシーケンス番号を指定できます。

要件

この操作には次の要件があります:

- ストリームマネージャー SDK の最小バージョン: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

例

次のスニペットは、`StreamName` という名前のストリームを更新します。Kinesis データストリームにエクスポートするストリームの複数プロパティを更新します。

Python

```
client = StreamManagerClient()

try:
    message_stream_info = client.describe_message_stream(STREAM_NAME)
    message_stream_info.definition.max_size=536870912
    message_stream_info.definition.stream_segment_size=33554432
    message_stream_info.definition.time_to_live_millis=3600000
    message_stream_info.definition.strategy_on_full=StrategyOnFull.RejectNewData
```

```
    message_stream_info.definition.persistence=Persistence.Memory
    message_stream_info.definition.flush_on_write=False
    message_stream_info.definition.export_definition.kinesis=
        [KinesisConfig(
            # Updating Export definition to add a Kinesis Stream configuration.
            identifier=str(uuid.uuid4()), kinesis_stream_name=str(uuid.uuid4()))]
    client.update_message_stream(message_stream_info.definition)
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: updateMessageStream | MessageStreamDefinition

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    MessageStreamInfo messageStreamInfo = client.describeMessageStream(STREAM_NAME);
    // Update the message stream with new values.
    client.updateMessageStream(
        messageStreamInfo.getDefinition()
            .withStrategyOnFull(StrategyOnFull.RejectNewData) // Required. Updating
 Strategy on full to reject new data.
            // Max Size update should be greater than initial Max Size defined in
 Create Message Stream request
            .withMaxSize(536870912L) // Update Max Size to 512 MB.
            .withStreamSegmentSize(33554432L) // Update Segment Size to 32 MB.
            .withFlushOnWrite(true) // Update flush on write to true.
            .withPersistence(Persistence.Memory) // Update the persistence to
 Memory.
            .withTimeToLiveMillis(3600000L)    // Update TTL to 1 hour.
            .withExportDefinition(
                // Optional. Choose where/how the stream is exported to the AWS ###
#.
                messageStreamInfo.getDefinition().getExportDefinition().
                    // Updating Export definition to add a Kinesis Stream
 configuration.
                    .withKinesis(new ArrayList<KinesisConfig>() {{
                        add(new KinesisConfig()
                            .withIdentifier(EXPORT_IDENTIFIER)
                            .withKinesisStreamName("test"));
```

```
                        }})
            );
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: update_message_stream | MessageStreamDefinition

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const messageStreamInfo = await c.describeMessageStream(STREAM_NAME);
        await client.updateMessageStream(
            messageStreamInfo.definition
                // Max Size update should be greater than initial Max Size defined
 in Create Message Stream request
                .withMaxSize(536870912)    // Default is 256 MB. Updating Max Size
 to 512 MB.
                .withStreamSegmentSize(33554432)    // Default is 16 MB. Updating
 Segment Size to 32 MB.
                .withTimeToLiveMillis(3600000)    // By default, no TTL is enabled.
 Update TTL to 1 hour.
                .withStrategyOnFull(StrategyOnFull.RejectNewData)    // Required.
 Updating Strategy on full to reject new data.
                .withPersistence(Persistence.Memory)    // Default is File. Update
 the persistence to Memory
                .withFlushOnWrite(true)    // Default is false. Updating to true.
                .withExportDefinition(
                    // Optional. Choose where/how the stream is exported to the AWS
 ####.
                    messageStreamInfo.definition.exportDefinition
                        // Updating Export definition to add a Kinesis Stream
 configuration.
                        .withKinesis([new
 KinesisConfig().withIdentifier(uuidv4()).withKinesisStreamName(uuidv4())])
                )
        );
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
```

```
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: updateMessageStream | MessageStreamDefinition

ストリーム更新の制約

ストリームを更新する際に次の制約が適用されます。次のリストに記載がない限り、更新は直ちに有効になります。

- ストリームの永続性を更新することはできません。この動作を変更するには、ストリームを削除して、新しい永続性ポリシーを定義するストリームを作成します。
- ストリームの最大サイズは、次の条件を満たしている場合に限って更新できます:
  - 最大サイズは、現在のストリームサイズ以上である必要があります。この情報を見つけるには、ストリームを記述して、返された MessageStreamInfo オブジェクトのストレージ状態を確認します。
  - 最大サイズは、ストリームのセグメントサイズ以上である必要があります。
- ストリーム セグメント サイズを、ストリームの最大サイズよりも小さい値に更新できます。更新した設定は、新しいセグメントに適用されます。
- 有効期限 (TTL) プロパティの更新は、新しい追加操作に適用されます。この値を減らす場合、ストリームマネージャーは TTL を超える既存のセグメントを削除することもあります。
- フルプロパティの戦略の更新は、新しい追加操作に適用されます。最も古いデータを上書きするように戦略を設定する場合、ストリームマネージャーは新しい設定に基づいて既存のセグメントも上書きすることがあります。
- 書き込み時のフラッシュのプロパティ更新は、新しいメッセージに適用されます。
- エクスポート設定の更新は、新しいエクスポートに適用されます。更新リクエストは、サポートするすべてのエクスポート設定を含める必要があります。それ以外の場合、ストリームマネージャーが削除します。
  - エクスポート設定を更新するとき、ターゲットのエクスポート設定の識別子を指定します。
  - エクスポート設定を追加するには、新しいエクスポート設定に対して一意の識別子を指定します。
  - エクスポート設定を削除するには、エクスポート設定を省略します。

- ストリームのエクスポート設定の開始シーケンス番号を更新するには、最後のシーケンス番号よりも小さい値を指定する必要があります。この情報を見つけるには、ストリームを記述して、返された MessageStreamInfo オブジェクトのストレージ状態を確認します。

メッセージストリームの削除

ストリームを削除します。ストリームを削除すると、ストリームに保存されているすべてのデータがディスクから削除されます。

要件

この操作には次の要件があります:

- ストリームマネージャー SDK の最小バージョン: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

例

次のスニペットは、StreamName という名前のストリームを削除します。

Python

```
client = StreamManagerClient()

try:
    client.delete_message_stream(stream_name="StreamName")
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: deleteMessageStream

Java

```
try (final StreamManagerClient client =
 StreamManagerClientFactory.standard().build()) {
    client.deleteMessageStream("StreamName");
} catch (StreamManagerException e) {
    // Properly handle exception.
```

```
}
```

Java SDK リファレンス: delete_message_stream

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        await client.deleteMessageStream("StreamName");
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: deleteMessageStream

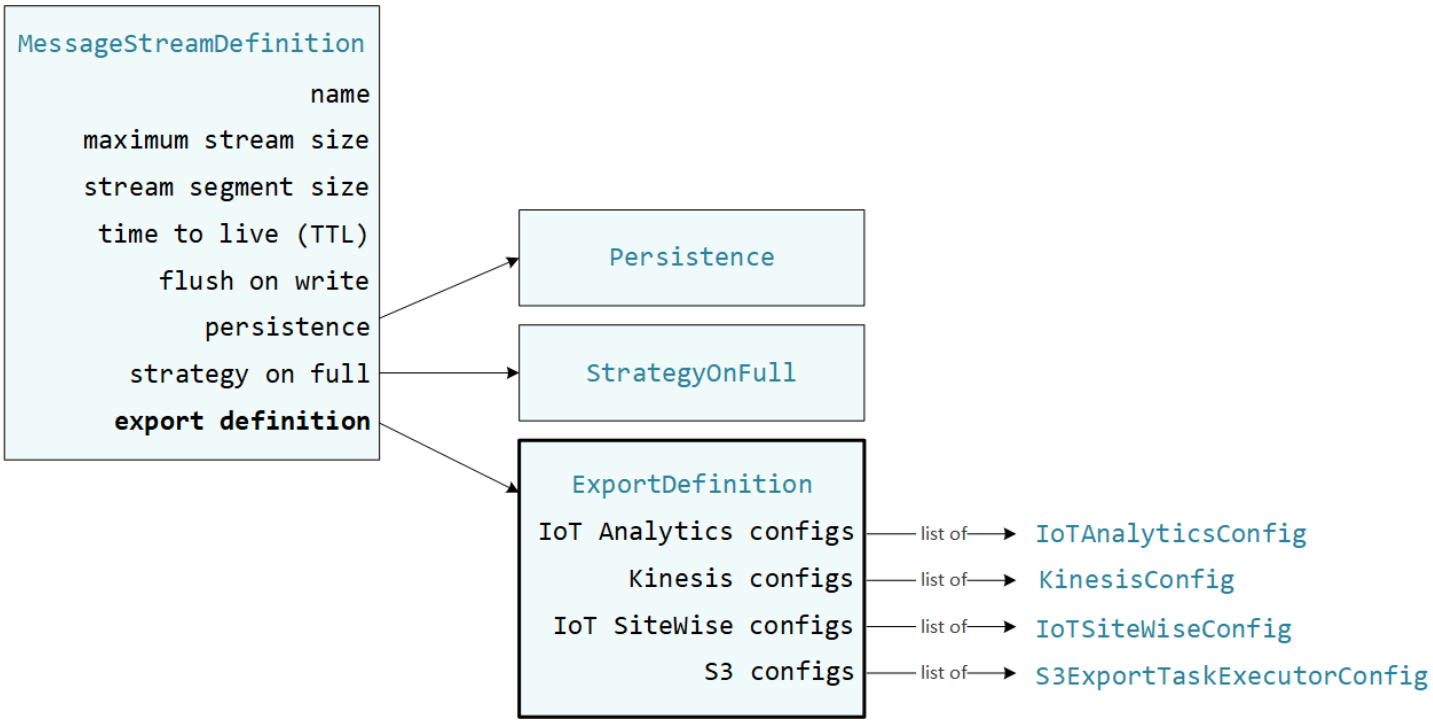
以下も参照してください。

- Greengrass コアデバイスでのデータストリームの管理
- AWS IoT Greengrass ストリームマネージャーの設定
- AWS クラウド でサポートされている送信先のエクスポート設定
- ストリームマネージャー SDK リファレンスの `StreamManagerClient`:
  - Python
  - Java
  - Node.js

AWS クラウド でサポートされている送信先のエクスポート設定

ユーザー定義の Greengrass コンポーネントは、ストリームマネージャー SDK で `StreamManagerClient` を使用して、ストリームマネージャーとやり取りします。コンポーネントは、[creates a stream] (ストリームの作成) または[updates a stream] (ストリームの更新) を行う際に、`MessageStreamDefinition` オブジェクト (エクスポート定義などのストリームプロパティ) を渡します。`ExportDefinition` オブジェクトには、そのストリームに定義されたエクスポート設

定が含まれています。ストリームマネージャーは、これらのエクスポート設定を使用して、ストリームをエクスポートする場所と方法を決定します。



1 つのストリームに 0 または 1 つ以上のエクスポートを定義できます。この場合、1 つの送信先タイプに複数のエクスポートを定義することも可能です。例えば、ストリームを 2 つの AWS IoT Analytics チャネルと 1 つの Kinesis データストリームにエクスポートできます。

エクスポートに失敗した場合、ストリームマネージャーは最大 5 分間隔で AWS クラウド へのデータのエクスポートを継続的に再試行します。再試行回数に上限はありません。

> **Note**
>
> `StreamManagerClient` を利用すると、ターゲットの送信先を使用して、ストリームを HTTP サーバーにエクスポートできます。このターゲットは、テストのみを目的としています。また、安定しておらず、実稼働環境での使用はサポートされていません。

サポート対象の AWS クラウド 送信先

- AWS IoT Analytics チャネル
- Amazon Kinesis Data Streams
- AWS IoT SiteWise アセットプロパティ
- Amazon S3 オブジェクト

これらの AWS クラウド リソースを維持する責任があります。

AWS IoT Analytics チャネル

ストリームマネージャーは、AWS IoT Analytics への自動エクスポートをサポートしています。AWS IoT Analytics による高度なデータ分析が、ビジネス上の意思決定や、機械学習モデルの改善に役立ちます。詳細については、「AWS IoT Analytics ユーザーガイド」の「AWS IoT Analytics とは?」を参照してください。

ストリームマネージャー SDK では、Greengrass コンポーネントは、`IoTAnalyticsConfig`を使用して、この宛先タイプのエクスポート設定を定義します。詳細については、ターゲット言語の SDK リファレンスを参照してください。

- Python SDK の IoTAnalyticsConfig
- Java SDK の IoTAnalyticsConfig
- Node.js SDK の IoTAnalyticsConfig

要件

このエクスポート先には以下の要件があります。

- AWS IoT Analytics のターゲットチャネルは、Greengrass コアデバイスと同じ AWS アカウントと AWS リージョン にある必要があります。
- コアデバイスが AWS サービスを操作できるように認証する は、ターゲットのチャネルに `iotanalytics:BatchPutMessage` 権限を許可する必要があります。例:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iotanalytics:BatchPutMessage"
      ],
      "Resource": [
        "arn:aws:iotanalytics:region:account-id:channel/channel_1_name",
        "arn:aws:iotanalytics:region:account-id:channel/channel_2_name"
      ]
    }
  ]
```

```
}
```

リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

AWS IoT Analytics にエクスポートする

AWS IoT Analytics にエクスポートするストリームを作成するには、Greengrass コンポーネントで 1 つ以上の `IoTAnalyticsConfig` オブジェクトを含むエクスポート定義を使用して [create a stream] (ストリームを作成) します。このオブジェクトによって、ターゲットチャネル、バッチサイズ、バッチ間隔、優先度などのエクスポート設定を定義します。

Greengrass コンポーネントがデバイスからデータを受信するとき、ターゲットストリームにデータの BLOB を含むメッセージを追加します。

その後、ストリームマネージャーは、ストリームのエクスポート設定で定義されたバッチ設定と優先度に基づいてデータをエクスポートします。

Amazon Kinesis Data Streams

ストリームマネージャーは、Amazon Kinesis Data Streams への自動エクスポートをサポートしています。Kinesis Data Streams は、大量のデータを集約し、データウェアハウスまたは MapReduce クラスターにロードするためによく使用されます。詳細については、「Amazon Kinesis デベロッパーガイド」の「Amazon Kinesis Data Streams とは」を参照してください。

ストリームマネージャー SDK では、Greengrass コンポーネントは、`KinesisConfig`を使用して、この宛先タイプのエクスポート設定を定義します。詳細については、ターゲット言語の SDK リファレンスを参照してください。

- KinesisConfig Python SDK の
- KinesisConfig Java SDK の
- KinesisConfig Node.js SDK の

要件

このエクスポート先には以下の要件があります。

- Kinesis Data Streams のターゲットストリームは、Greengrass コアデバイスと同じ AWS アカウント と AWS リージョン にある必要があります。
- コアデバイスが AWS サービスを操作できるように認証する は、ターゲットデータストリームに `kinesis:PutRecords` 権限を許可する必要があります。例:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "kinesis:PutRecords"
      ],
      "Resource": [
        "arn:aws:kinesis:region:account-id:stream/stream_1_name",
        "arn:aws:kinesis:region:account-id:stream/stream_2_name"
      ]
    }
  ]
}
```

  リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

Kinesis Data Streams へのエクスポート

Kinesis Data Streams にエクスポートするストリームを作成するには、Greengrass コンポーネントで 1 つ以上の `KinesisConfig` オブジェクトを含むエクスポート定義を使用して [create a stream] (ストリームを作成) します。このオブジェクトによって、ターゲットデータストリーム、バッチサイズ、バッチ間隔、優先度などのエクスポート設定を定義します。

Greengrass コンポーネントがデバイスからデータを受信するとき、ターゲットストリームにデータの BLOB を含むメッセージを追加します。その後、ストリームマネージャーは、ストリームのエクスポート設定で定義されたバッチ設定と優先度に基づいてデータをエクスポートします。

ストリームマネージャーは、Amazon Kinesis にアップロードされた各レコードのパーティションキーとして、一意のランダムな UUID を生成します。

AWS IoT SiteWise アセットプロパティ

ストリームマネージャーは、AWS IoT SiteWise への自動エクスポートをサポートしています。AWS IoT SiteWiseを使用すると、産業機器からのデータを大規模に収集、整理、分析できます。詳細については、「AWS IoT SiteWise ユーザーガイド」の「AWS IoT SiteWise とは?」を参照してください。

ストリームマネージャー SDK では、Greengrass コンポーネントは、`IoTSiteWiseConfig`を使用して、この宛先タイプのエクスポート設定を定義します。詳細については、ターゲット言語の SDK リファレンスを参照してください。

- Python SDK の IoTSiteWiseConfig
- Java SDK の IoTSiteWiseConfig
- Node.js SDK の IoTSiteWiseConfig

> **Note**
>
> AWS には、AWS IoT SiteWise コンポーネントも用意されており、OPC-UA ソースからのデータストリーミングに使用できるあらかじめ構築されたソリューションを提供します。詳細については、「IoT SiteWise OPC-UA コレクター」を参照してください。

要件

このエクスポート先には以下の要件があります。

- AWS IoT SiteWise のターゲットアセットプロパティは、Greengrass コアデバイスと同じ AWS アカウント と AWS リージョン にある必要があります。

  > **Note**
  >
  > AWS IoT SiteWise がサポートする AWS リージョン のリストについては、「AWS 全般のリファレンス」の「AWS IoT SiteWise エンドポイントとクォータ」を参照してください。

- コアデバイスが AWS サービスを操作できるように認証する は、ターゲットのアセットプロパティに `iotsitewise:BatchPutAssetPropertyValue` 権限を許可する必要があります。次の例では、ポリシーで `iotsitewise:assetHierarchyPath` 条件キーを使用して、ターゲットルートアセットとその子へのアクセスを許可しています。ポリシーから `Condition` を削除して、

すべての AWS IoT SiteWise アセットへのアクセスを許可したり、個々のアセットの ARN を指定したりできます。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
       "Effect": "Allow",
       "Action": "iotsitewise:BatchPutAssetPropertyValue",
       "Resource": "*",
       "Condition": {
         "StringLike": {
           "iotsitewise:assetHierarchyPath": [
             "/root node asset ID",
             "/root node asset ID/*"
           ]
         }
       }
    }
  ]
}
```

リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

重要なセキュリティ情報については、「 ユーザーガイド」の「 BatchPutAssetPropertyValue 認可」を参照してください。 AWS IoT SiteWise

AWS IoT SiteWise にエクスポートする

AWS IoT SiteWise にエクスポートするストリームを作成するには、Greengrass コンポーネントで 1 つ以上の `IoTSiteWiseConfig` オブジェクトを含むエクスポート定義を使用して [create a stream] (ストリームを作成) します。このオブジェクトによって、バッチサイズ、バッチ間隔、優先度などのエクスポート設定を定義します。

Greengrass コンポーネントがデバイスからアセットプロパティデータを受信するとき、ターゲットストリームにデータを含むメッセージを追加します。メッセージは、JSON シリアル化された `PutAssetPropertyValueEntry` オブジェクトであり、これには、1 つ以上のアセットプロパティ

に対するプロパティ値が含まれています。詳細については、AWS IoT SiteWise のエクスポート先に関する「メッセージの追加」を参照してください。

> **Note**
>
> AWS IoT SiteWise にデータを送信する場合、データは `BatchPutAssetPropertyValue` アクションの要件を満たしている必要があります。詳細については、AWS IoT SiteWise API リファレンスの「BatchPutAssetPropertyValue」を参照してください。

その後、ストリームマネージャーは、ストリームのエクスポート設定で定義されたバッチ設定と優先度に基づいてデータをエクスポートします。

ストリームマネージャーの設定と Greengrass コンポーネントロジックを調整して、エクスポート対策を設計できます。例:

- ほぼリアルタイムのエクスポートでは、少ないバッチサイズと短い間隔を設定して、受信したデータをストリームに追加します。
- バッチ処理を最適化し、帯域幅の制約を軽減し、コストを最小限に抑えるために、Greengrass コンポーネントは、データをストリームに追加する前に、単一のアセットプロパティに対して受信した timestamp-quality-value (TQV) データポイントをプールできます。対策を 1 つ挙げるとすれば、それは、同じプロパティのエントリを複数送信するのではなく、最大 10 の異なるプロパティとアセットの組み合わせ、またはプロパティエイリアスのエントリを 1 つのメッセージでバッチ処理することです。これにより、ストリームマネージャーは AWS IoT SiteWise クォータ内にとどまることができます。

Amazon S3 オブジェクト

ストリームマネージャーは、Amazon S3 への自動エクスポートをサポートしています。Amazon S3 を使用すると、膨大なデータの保存と取得を行えます。詳細については、「Amazon Simple Storage Service デベロッパーガイド」の「Amazon S3 とは」を参照してください。

ストリームマネージャー SDK では、Greengrass コンポーネントは、`S3ExportTaskExecutorConfig`を使用して、この宛先タイプのエクスポート設定を定義します。詳細については、ターゲット言語の SDK リファレンスを参照してください。

- Python SDK の S3ExportTaskExecutorConfig
- Java SDK の S3ExportTaskExecutorConfig

- Node.js SDK の S3ExportTaskExecutorConfig

要件

このエクスポート先には以下の要件があります。

- ターゲット Amazon S3 バケットは Greengrass コアデバイスと同じ AWS アカウント にある必要があります。
- [Greengrass container] (Greengrass コンテナ) モードで実行される Lambda 関数が入力ファイルを入力ファイルディレクトリに書き込む場合は、書き込み権限を持つコンテナ内のボリュームとしてディレクトリをマウントする必要があります。これにより、ファイルがルートファイルシステムに書き込まれ、コンテナの外部で実行されるストリームマネージャーコンポーネントに表示されるようになります。
- Docker コンテナコンポーネントが入力ファイルを入力ファイルディレクトリに書き込む場合は、書き込み権限を持つコンテナ内のボリュームとしてディレクトリをマウントする必要があります。これにより、ファイルがルートファイルシステムに書き込まれ、コンテナの外部で実行されるストリームマネージャーコンポーネントに表示されるようになります。
- コアデバイスが AWS サービスを操作できるように認証する で、ターゲットのバケットに次の権限を許可する必要があります。例:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:AbortMultipartUpload",
        "s3:ListMultipartUploadParts"
      ],
      "Resource": [
        "arn:aws:s3:::bucket-1-name/*",
        "arn:aws:s3:::bucket-2-name/*"
      ]
    }
  ]
}
```

リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

Amazon S3 へのエクスポート

Amazon S3 にエクスポートするストリームを作成するには、Greengrass コンポーネントで `S3ExportTaskExecutorConfig` オブジェクトを使用してエクスポートポリシーを設定します。このポリシーによって、マルチパートアップロードのしきい値や優先度といったエクスポート設定を定義します。Amazon S3 エクスポートでは、ストリームマネージャーが、コアデバイス上のローカルファイルから読み取るデータをアップロードします。アップロードを開始するには、Greengrass コンポーネントでエクスポートタスクをターゲットストリームに追加します。エクスポートタスクには、入力ファイルとターゲット Amazon S3 オブジェクトの情報が含まれます。ストリームマネージャーは、ストリームに追加された順にタスクを実行します。

> **Note**
>
> ターゲットバケットは、AWS アカウント に存在しなければなりません。指定したキーのオブジェクトが存在しない場合は、ストリームマネージャーによってオブジェクトが作成されます。

ストリームマネージャーは、マルチパートアップロードしきい値プロパティ、最小パーツサイズ設定、入力ファイルサイズを使用して、データのアップロード方法を決定します。マルチパートアップロードのしきい値は、最小パートサイズ以上でなければなりません。データを並行してアップロードする場合は、複数のストリームを作成できます。

ターゲット Amazon S3 オブジェクトを指定するキーには、`!{timestamp:value}`有効な Java DateTimeFormatter 文字列をプレースホルダーに含めることができます。これらのタイムスタンププレースホルダーを使用すると、入力ファイルデータがアップロードされた時刻に基づいて Amazon S3 のデータを分割できます。例えば、次のキー名は、`my-key/2020/12/31/data.txt` などの値に解決されます。

```
my-key/!{timestamp:YYYY}/!{timestamp:MM}/!{timestamp:dd}/data.txt
```

> **Note**
>
> ストリームのエクスポートステータスを監視する場合は、まずステータスストリームを作成して、そのストリームを使用するようにエクスポートストリームを設定します。詳細については、「the section called “エクスポートタスクの監視”」を参照してください。

入力データの管理

IoT アプリケーションが入力データのライフサイクル管理に使用するコードを作成できます。次のワークフロー例は、Greengrass コンポーネントを使用してこのデータを管理する方法を示しています。

1. ローカルプロセスは、デバイスまたは周辺機器からデータを受信し、コアデバイスのディレクトリ内にあるファイルにデータを書き込みます。これらが、ストリームマネージャーの入力ファイルとなります。
2. Greengrass コンポーネントはディレクトリをスキャンし、新しいファイルが作成されると、ターゲットストリームに[appends an export task] (エクスポートタスクを追加) します。このタスクは、JSON シリアル化された `S3ExportTaskDefinition` オブジェクトであり、これによって、入力ファイルの URL、ターゲットの Amazon S3 バケットとキー、オプションのユーザーメタデータが指定されます。
3. ストリームマネージャーは、入力ファイルを読み取り、追加されたタスクの順に Amazon S3 にデータをエクスポートします。ターゲットバケットは、AWS アカウント に存在しなければなりません。指定したキーのオブジェクトが存在しない場合は、ストリームマネージャーによってオブジェクトが作成されます。
4. Greengrass コンポーネントは、ステータスストリームから[reads messages] (メッセージを読み取り)、エクスポートステータスを監視します。エクスポートタスクが完了すると、Greengrass コンポーネントは対応する入力ファイルを削除します。詳細については、「the section called “エクスポートタスクの監視”」を参照してください。

エクスポートタスクの監視

IoT アプリケーションが Amazon S3 エクスポートのステータス監視に使用するコードを作成できます。Greengrass コンポーネントは、ステータスストリームを作成して、そのストリームにステータス更新を書き込むようにエクスポートストリームを設定する必要があります。1 つのステータスストリームは、Amazon S3 にエクスポートする複数のストリームからステータスの更新を受け取ることができます。

まず、ステータスストリームとして使用するストリームを作成します。ストリームのサイズとリテンションポリシーを設定して、ステータスメッセージのライフスパンを制御できます。例:

- ステータスメッセージを保存しない場合は、`Persistence` を `Memory` に設定します。
- 新しいステータスメッセージが失われないようにするには、`StrategyOnFull` を `OverwriteOldestData` に設定します。

次に、ステータスストリームを使用するようにエクスポートストリームを作成または更新します。具体的には、ストリームの `S3ExportTaskExecutorConfig` エクスポート設定のステータス設定プロパティを設定します。この設定により、エクスポートタスクに関するステータスメッセージをステータスストリームに書き込むようにストリームマネージャーに指示します。`StatusConfig` オブジェクトで、ステータスストリームの名前と冗長性のレベルを指定します。サポート対象の値を次に示します。最も冗長でないもの (`ERROR`) から最も冗長なもの (`TRACE`) を表しています。デフォルトは `INFO` です。

- `ERROR`
- `WARN`
- `INFO`
- `DEBUG`
- `TRACE`

次のワークフロー例は、Greengrass コンポーネントがステータスストリームを使用してエクスポートステータスを監視する方法を示しています。

1. 前のワークフローで説明したように、Greengrass コンポーネントは、エクスポートタスクに関するステータスメッセージをステータスストリームに書き込むように設定されたストリームに [appends an export task] (エクスポートタスクを追加) します。この追加の操作によって、タスク ID を表すシーケンス番号が返ります。
2. Greengrass コンポーネントは、ステータスストリームから順番に[reads messages] (メッセージを読み取り) ます。その後、ストリーム名とタスク ID に基づいて、またはメッセージコンテキストからのエクスポートタスクプロパティに基づいてメッセージをフィルタリングします。例えば、Greengrass コンポーネントは、エクスポートタスクの入力ファイル URL でフィルタリングできます。このタスクは、メッセージコンテキストの `S3ExportTaskDefinition` オブジェクトで表されます。

   次のステータスコードは、エクスポートタスクが完了の状態になったことを示します。

- `Success`。アップロードは正常に完了しました。
- `Failure`。ストリームマネージャーでエラー (例: 指定したバケットが存在しないなど) が発生しました。問題の解決後に、エクスポートタスクをストリームに再度追加できます。
- `Canceled`。 ストリームまたはエクスポート定義が削除されたか、タスクの time-to-live (TTL) 期間が終了したため、タスクは停止されました。

> **Note**
>
> タスクのステータスは `InProgress` または `Warning` の場合もあります。ストリームマネージャーは、タスクの実行に影響しないエラーがイベントから返ったときに警告を発行します。例えば、部分的アップロードのクリーンアップが失敗すると、警告を返します。

3. エクスポートタスクが完了すると、Greengrass コンポーネントは対応する入力ファイルを削除します。

次の例は、Greengrass コンポーネントがステータスメッセージを読み取り、処理する方法を示しています。

Python

```
import time
from stream_manager import (
    ReadMessagesOptions,
    Status,
    StatusConfig,
    StatusLevel,
    StatusMessage,
    StreamManagerClient,
)
from stream_manager.util import Util

client = StreamManagerClient()

try:
    # Read the statuses from the export status stream
    is_file_uploaded_to_s3 = False
    while not is_file_uploaded_to_s3:
        try:
            messages_list = client.read_messages(
```

```
                "StatusStreamName", ReadMessagesOptions(min_message_count=1,
 read_timeout_millis=1000)
            )
            for message in messages_list:
                # Deserialize the status message first.
                status_message = Util.deserialize_json_bytes_to_obj(message.payload,
 StatusMessage)

                # Check the status of the status message. If the status is
 "Success",
                # the file was successfully uploaded to S3.
                # If the status was either "Failure" or "Cancelled", the server was
 unable to upload the file to S3.
                # We will print the message for why the upload to S3 failed from the
 status message.
                # If the status was "InProgress", the status indicates that the
 server has started uploading
                # the S3 task.
                if status_message.status == Status.Success:
                    logger.info("Successfully uploaded file at path " + file_url + "
 to S3.")
                    is_file_uploaded_to_s3 = True
                elif status_message.status == Status.Failure or
 status_message.status == Status.Canceled:
                    logger.info(
                        "Unable to upload file at path " + file_url + " to S3.
 Message: " + status_message.message
                    )
                    is_file_uploaded_to_s3 = True
            time.sleep(5)
        except StreamManagerException:
            logger.exception("Exception while running")
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: read_messages | StatusMessage

Java

```
import com.amazonaws.greengrass.streammanager.client.StreamManagerClient;
```

```
import com.amazonaws.greengrass.streammanager.client.StreamManagerClientFactory;
import com.amazonaws.greengrass.streammanager.client.utils.ValidateAndSerialize;
import com.amazonaws.greengrass.streammanager.model.ReadMessagesOptions;
import com.amazonaws.greengrass.streammanager.model.Status;
import com.amazonaws.greengrass.streammanager.model.StatusConfig;
import com.amazonaws.greengrass.streammanager.model.StatusLevel;
import com.amazonaws.greengrass.streammanager.model.StatusMessage;

 try (final StreamManagerClient client =
 StreamManagerClientFactory.standard().build()) {
    try {
        boolean isS3UploadComplete = false;
        while (!isS3UploadComplete) {
            try {
                // Read the statuses from the export status stream
                List<Message> messages = client.readMessages("StatusStreamName",
                    new
 ReadMessagesOptions().withMinMessageCount(1L).withReadTimeoutMillis(1000L));
                for (Message message : messages) {
                    // Deserialize the status message first.
                    StatusMessage statusMessage =
 ValidateAndSerialize.deserializeJsonBytesToObj(message.getPayload(),
 StatusMessage.class);
                    // Check the status of the status message. If the status is
 "Success", the file was successfully uploaded to S3.
                    // If the status was either "Failure" or "Canceled", the server
 was unable to upload the file to S3.
                    // We will print the message for why the upload to S3 failed
 from the status message.
                    // If the status was "InProgress", the status indicates that the
 server has started uploading the S3 task.
                    if (Status.Success.equals(statusMessage.getStatus())) {
                        System.out.println("Successfully uploaded file at path " +
 FILE_URL + " to S3.");
                        isS3UploadComplete = true;
                     } else if (Status.Failure.equals(statusMessage.getStatus()) ||
 Status.Canceled.equals(statusMessage.getStatus())) {
                        System.out.println(String.format("Unable to upload file at
 path %s to S3. Message %s",

 statusMessage.getStatusContext().getS3ExportTaskDefinition().getInputUrl(),
                            statusMessage.getMessage()));
                        sS3UploadComplete = true;
                    }
```

```
                }
            } catch (StreamManagerException ignored) {
            } finally {
                // Sleep for sometime for the S3 upload task to complete before
 trying to read the status message.
                Thread.sleep(5000);
            }
        } catch (e) {
        // Properly handle errors.
    }
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: readMessages | StatusMessage

Node.js

```
const {
    StreamManagerClient, ReadMessagesOptions,
    Status, StatusConfig, StatusLevel, StatusMessage,
    util,
} = require(*'aws-greengrass-stream-manager-sdk'*);

const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        let isS3UploadComplete = false;
        while (!isS3UploadComplete) {
            try {
                // Read the statuses from the export status stream
                const messages = await c.readMessages("StatusStreamName",
                    new ReadMessagesOptions()
                        .withMinMessageCount(1)
                        .withReadTimeoutMillis(1000));

                messages.forEach((message) => {
                    // Deserialize the status message first.
                    const statusMessage =
 util.deserializeJsonBytesToObj(message.payload, StatusMessage);
                    // Check the status of the status message. If the status is
 'Success', the file was successfully uploaded to S3.
                    // If the status was either 'Failure' or 'Cancelled', the server
 was unable to upload the file to S3.
```

```
                    // We will print the message for why the upload to S3 failed
 from the status message.
                    // If the status was "InProgress", the status indicates that the
 server has started uploading the S3 task.
                    if (statusMessage.status === Status.Success) {
                        console.log(`Successfully uploaded file at path ${FILE_URL}
 to S3.`);
                        isS3UploadComplete = true;
                    } else if (statusMessage.status === Status.Failure ||
 statusMessage.status === Status.Canceled) {
                        console.log(`Unable to upload file at path ${FILE_URL} to
 S3. Message: ${statusMessage.message}`);
                        isS3UploadComplete = true;
                    }
                });
                // Sleep for sometime for the S3 upload task to complete before
 trying to read the status message.
                await new Promise((r) => setTimeout(r, 5000));
            } catch (e) {
                // Ignored
            }
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: readMessages | StatusMessage

AWS IoT Greengrass ストリームマネージャーの設定

Greengrass コアデバイスでは、ストリームマネージャーは IoT デバイスのデータを保存、処理、およびエクスポートすることができます。ストリームマネージャーは、ランタイム設定の設定に使用するパラメータを提供します。これらの設定は、Greengrass コアデバイスのすべてのストリームに適用されます。コンポーネントをデプロイする際に、AWS IoT Greengrass コンソールまたは API を使用して、ストリームマネージャーの設定を行うことができます。変更は、デプロイを完了した後に有効になります。

ストリームマネージャーのパラメータ

ストリームマネージャーは、コンポーネントをコアデバイスにデプロイする際に設定可能な以下のパラメータを提供します。すべてのパラメータは省略可能です。

[Storage directory] (ストレージディレクトリ)

パラメータ名: `STREAM_MANAGER_STORE_ROOT_DIR`

ストリームを保存するために使用されるローカルフォルダの絶対パス。この値は、スラッシュ (`/data` など) で開始する必要があります。

既存のフォルダを指定する必要があります。ストリームマネージャーコンポーネントを実行するシステムユーザーには、このフォルダに対する読み取りと書き込み許可が必要です。例えば、次のコマンドを実行して、ストリームマネージャーのルートフォルダとして指定するフォルダ `/var/greengrass/streams` を作成および設定できます。これらのコマンドは、デフォルトのシステムユーザーである `ggc_user` が、このフォルダを読み取りおよび書き込みできるようにします。

```
sudo mkdir /var/greengrass/streams
sudo chown ggc_user /var/greengrass/streams
sudo chmod 700 /var/greengrass/streams
```

ストリームデータのセキュリティ保護については、「the section called “ローカルデータセキュリティ”」を参照してください。

デフォルト: *`/greengrass/v2`*`/work/aws.greengrass.StreamManager`

Server port

パラメータ名: `STREAM_MANAGER_SERVER_PORT`

ストリームマネージャーとの通信に使用されるローカルポート番号。デフォルトは `8088` です。

`0` を指定して、ランダムに利用可能なポートを利用できます。

クライアントを認証する

パラメータ名: `STREAM_MANAGER_AUTHENTICATE_CLIENT`

ストリームマネージャーと対話するためにクライアントを認証する必要があるかどうかを示します。クライアントとストリームマネージャー間のすべてのやり取りは、ストリームマネージャー

SDK によって制御されます。このパラメータは、ストリームを操作するためにストリームマネージャー SDK を呼び出すことができるクライアントを決定します。詳細については、「the section called “クライアント承認”」を参照してください。

有効な値は `true` または `false` です。デフォルトは `true` (推奨) です。

- `true`。Greengrass コンポーネントのみをクライアントとして許可します。コンポーネントは内部 AWS IoT Greengrass Core プロトコルを使用して、ストリームマネージャー SDK で認証します。
- `false`。AWS IoT Greengrass Core で実行されるすべてのプロセスをクライアントとして許可します。ビジネスケースで必要とされない限り、値を `false` に設定しないでください。例えば、コアデバイス上の非コンポーネントプロセスがストリームマネージャーと直接通信する必要がある場合に限って `false` を使用します。

最大帯域幅

パラメータ名: `STREAM_MANAGER_EXPORTER_MAX_BANDWIDTH`

データのエクスポートに使用できる平均最大帯域幅 (キロビット/秒)。デフォルトでは、使用可能な帯域幅を無制限に使用することができます。

スレッドプールサイズ

パラメータ名: `STREAM_MANAGER_EXPORTER_THREAD_POOL_SIZE`

データのエクスポートに使用できるアクティブなスレッドの最大数。デフォルトは `5` です。

最適なサイズは、ハードウェア、ストリームボリューム、予定されているエクスポートストリームの数によって異なります。エクスポート速度が遅い場合は、この設定を調整して、ハードウェアとビジネスケースに最適なサイズを見つけることができます。コアデバイスハードウェアの CPU とメモリは、制限要因です。開始するには、この値をデバイスのプロセッサコアの数と同じ値に設定してみてください。

ハードウェアがサポートできるサイズよりも大きいサイズを設定しないように注意してください。各ストリームはハードウェアリソースを消費するため、制約のあるデバイス上ではエクスポートストリームの数を制限する必要があります。

JVM の引数

パラメータ名: `JVM_ARGS`

起動時にストリームマネージャーに渡すカスタム Java 仮想マシン引数。複数の引数はスペースで区切る必要があります。

このパラメータは、JVM で使用されるデフォルト設定を上書きする必要がある場合にのみ使用します。例えば、大量のストリームをエクスポートする場合は、デフォルトのヒープサイズを大きくする必要があります。

ログ記録レベル

パラメータ名: `LOG_LEVEL`

コンポーネントのロギングレベル。こちらにレベル順に一覧表示されているログレベルから選択します。

- `TRACE`
- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

デフォルト: `INFO`

マルチパートアップロードの最小サイズ

パラメータ名: `STREAM_MANAGER_EXPORTER_S3_DESTINATION_MULTIPART_UPLOAD_MIN_PART_SIZE_BYTES`

Amazon S3 へのマルチパートアップロードにおけるパートの最小サイズ (バイト単位)。ストリームマネージャーはこの設定と入力ファイルのサイズを基に、マルチパート PUT リクエストのデータをバッチ処理する方法を決定します。デフォルトの最小値は `5242880` バイト (5 MB) です。

> **Note**
>
> ストリームマネージャーはストリームの `sizeThresholdForMultipartUploadBytes` プロパティを基に、Amazon S3 へのエクスポートをシングルアップロードで行うか、マルチパートアップロードで行うかを決定します。ユーザー定義の Greengrass コンポーネントが、Amazon S3 にエクスポートするストリームを作成する際にこのしきい値を設定します。デフォルトのしきい値は 5 MB です。

以下も参照してください。

- Greengrass コアデバイスでのデータストリームの管理

- StreamManagerClient を使用してストリームを操作する
- AWS クラウド でサポートされている送信先のエクスポート設定

機械学習の推論を実行する

AWS IoT Greengrass で、クラウドトレーニング済みモデルを使用し、ローカルに生成されたデータに対して、エッジデバイスに機械学習 (ML) の推論を実行できます。ローカル推論の実行により低レイテンシーとコスト節約のメリットを得ながら、モデルのトレーニングと複雑な処理にクラウドコンピューティングの処理能力を活用できます。

AWS IoT Greengrass は、推論をより効率的に実行するために必要な手順を作成します。推論モデルをどこでもトレーニングし、機械学習コンポーネントとしてローカルにデプロイできます。例えば、Amazon SageMaker で深層学習モデルを構築してトレーニングしたり、Amazon Lookout for Vision でコンピュータビジョンモデルを構築してトレーニングしたりできます。次に、コンポーネントでこれらのモデルをアーティファクトとして使用してコアデバイスに推論を実行するため、これらのモデルを Amazon S3 バケットに保存できます。

トピック

AWS IoT Greengrass ML 推論のしくみ

AWS は、デバイスに機械学習の推論を実行するワンステップデプロイの作成に使用できる機械学習コンポーネントを提供します。これらのコンポーネントをテンプレートとしても使用し、お客様の特定の要件を満たすカスタムコンポーネントを作成できます。

AWS は、次の機械学習コンポーネントのカテゴリを示します。

- モデルコンポーネント - Greengrass アーティファクトとして機械学習モデルが含まれます。
- ランタイムコンポーネント - 機械学習フレームワークとその従属関係を Greengrass コアデバイスにインストールするスクリプトが含まれます。
- 推論コンポーネント - 推論コードが含まれ、機械学習フレームワークをインストールして事前学習済みの機械学習モデルをダウンロードするためのコンポーネント従属関係が含まれます。

機械学習の推論を実行するために作成する各デプロイは、推論アプリケーションの実行、機械学習フレームワークのインストール、機械学習モデルのダウンロードをするコンポーネントが少なくとも 1 つ設定されています。AWS が提供するコンポーネントでサンプル推論を実行するには、推論コンポーネントをコアデバイスにデプロイします。コアデバイスは、対応するモデルとランタイムコンポーネントを従属関係として、自動的に含めます。デプロイをカスタマイズするには、カスタムモデルコンポーネントでサンプルモデルコンポーネントをプラグインまたはスワップアウトしたり、テンプレートとして AWS が提供するコンポーネントにコンポーネントレシピを使用して独自のカスタム推論、モデル、ランタイムコンポーネントを作成したりできます。

カスタムコンポーネントを使用して機械学習の推論を実行するには、次の手順を実行します。

1. モデルコンポーネントを作成します。このコンポーネントには、推論の実行に使用する機械学習モデルが含まれています。 AWSには、事前トレーニング済みの DLR モデルと TensorFlow Lite モデルのサンプルが用意されています。カスタムモデルを使用するには、独自のモデルコンポーネントを作成します。
2. ランタイムコンポーネントを作成します。このコンポーネントには、モデルの機械学習ランタイムをインストールするために必要なスクリプトが含まれています。 AWSには、深層学習ランタイム (DLR) と TensorFlow Lite のサンプルランタイムコンポーネントが用意されています。カスタムモデルと推論コードで他のランタイムを使用するには、独自のランタイムコンポーネントを作成します。
3. 推論コンポーネントを作成します。このコンポーネントには推論コードが含まれており、モデルコンポーネントとランタイムコンポーネントが依存関係として含まれています。 AWSは、DLR と TensorFlow Lite を使用したイメージ分類とオブジェクト検出のサンプル推論コンポーネントを提供します。他のタイプの推論を実行したり、カスタムモデルやランタイムを使用したりするには、独自の推論コンポーネントを作成します。
4. 推論コンポーネントをデプロイします。このコンポーネントをデプロイするとき、AWS IoT Greengrass はモデルとランタイムコンポーネントの従属関係を自動的にデプロイもします。

AWS が提供するコンポーネントの使用を開始するには、「the section called “サンプルイメージ分類推論の実行”」を参照してください。

カスタム機械学習コンポーネントの作成方法については、「機械学習コンポーネントのカスタマイズ」を参照してください。

AWS IoT Greengrass バージョン 2 との相違点

AWS IoT Greengrass は、モデル、ランタイム、推論コードなど、機械学習の機能ユニットをコンポーネントに統合し、機械学習ランタイムのインストール、トレーニング済モデルのダウンロード、デバイスに推論を実行するため、ワンステッププロセスを使用できるようにします。

AWS が提供する機械学習コンポーネントを使用することにより、サンプル推論コードと事前トレーニング済みモデルで機械学習の推論の実行を開始できる柔軟性が得られます。カスタムモデルコンポーネントをプラグインして、AWS が提供する推論とランタイムコンポーネントと一緒に独自のカスタムトレーニング済モデルを使用できます。完全にカスタマイズされた機械学習ソリューションの場合、パブリックコンポーネントをテンプレートとして使用してカスタムコンポーネントを作成し、任意のランタイム、モデル、推論タイプを使用できます。

要件

機械学習コンポーネントを作成して使用するには、次のものが必要になります。

- Greengrass コアデバイス。アカウントをお持ちでない場合は、「チュートリアル: AWS IoT Greengrass V2 の開始方法」を参照してください。
- AWS が提供するサンプル機械学習コンポーネントを使用するために最低 500 MB のローカルのストレージ容量。

サポートされているモデルソース

AWS IoT Greengrass は、Amazon S3 に保存されるカスタムトレーニング済機械学習モデルの使用をサポートしています。Amazon SageMaker エッジパッケージングジョブを使用して、SageMaker Neo コンパイルモデルのモデルコンポーネントを直接作成することもできます。で SageMaker Edge Manager を使用する方法についてはAWS IoT Greengrass、「」を参照してくださいGreengrass コアデバイスで Amazon SageMaker Edge Manager を使用する。Amazon Lookout for Vision のモデルパッケージングジョブを使用して、Lookout for Vision モデルのモデルコンポーネントを作成することもできます。Lookout for Vision を AWS IoT Greengrass と使用する方法の詳細については、「Greengrass コアデバイスのAmazon Lookout for Vision を使用」を参照してください。

モデルを含む S3 バケットは、次の要件を満たしている必要があります:

- SSE-C を使用して暗号化してはなりません。サーバー側の暗号化を使用するバケットの場合、AWS IoT Greengrass 機械学習の推論は現在、SSE-S3 または SSE-KMS 暗号化オプションのみをサポートしています。サーバー側の暗号化の詳細については、「Amazon Simple Storage Service ユーザーガイド」の「サーバー側の暗号化を使用したデータ保護」を参照してください。
- それらの名前にピリオド (.) を含めることはできません。詳細については、「Amazon Simple Storage Service ユーザーガイド」のバケット命名規則で、SSL で仮想ホスト型バケットの使用に関するルールを参照してください。
- モデルソースを保存する S3 バケットは、機械学習コンポーネントと同じ AWS アカウント と AWS リージョン にある必要があります。
- AWS IoT Greengrass はモデルソースに対する `read` 許可が必要になります。AWS IoT Greengrass に対して S3 バケットへのアクセスを有効化するには、Greengrass デバイスのロールは `s3:GetObject` アクションを許可する必要があります。デバイスロールの詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

サポートされている機械学習ランタイム

AWS IoT Greengrass は、カスタムトレーニング済モデルで機械学習の推論を実行するため、カスタムコンポーネントを作成して、任意の機械学習ランタイムを使用できます。カスタム機械学習コンポーネントの作成方法については、「機械学習コンポーネントのカスタマイズ」を参照してください。

機械学習を始めるプロセスの効率を向上させるため、AWS IoT Greengrass は、次の機械学習ランタイムを使用する推論、モデル、ランタイムコンポーネントのサンプルを示します。

- 深層学習ランタイム (DLR) v1.6.0 と v1.3.0
- TensorFlow Lite v2.5.0

AWS が提供する機械学習コンポーネント

次の表では、機械学習に使用される AWS が提供するコンポーネントを一覧表示します。

> **Note**
>
> AWS が提供する複数のコンポーネントは、Greengrass nucleus の特定マイナーバージョンに依存します。この従属関係により、Greengrass nucleus を新しいマイナーバージョンに更新するとき、これらのコンポーネントを更新する必要があります。各コンポーネントが依

存する nucleus の特定バージョンの情報については、対応するコンポーネントのトピックを参照してください。nucleus の更新の詳細については、「AWS IoT Greengrass Core ソフトウェア (OTA) の更新」を参照してください。

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
Lookout for Vision エッジエージェント	Amazon Lookout for Vision ランタイムを Greengrass コアデバイスにデプロイし、コンピュータビジョンを使用して産業製品の欠陥を検出できるようにします。	ジェネリック	Linux	いいえ
SageMaker Edge マネージャー	Amazon SageMaker Edge Manager エージェントを Greengrass コアデバイスにデプロイします。	ジェネリック	Linux、Windows	いいえ
DLR イメージ分類	DLR イメージ分類モデルストアと DLR ランタイムコンポーネントを従属	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
	関係として使用し、DLR をインストール、サンプルイメージ分類モデルをダウンロード、サポートされているデバイスにイメージ分類推論を実行する推論コンポーネント。			
DLR オブジェクトの検出	DLR オブジェクト検知モデルストアと DLR ランタイムコンポーネントを従属関係として使用し、DLR をインストール、サンプルオブジェクト検知モデルをダウンロード、サポートされているデバイスにオブジェクト検知推論を実行する推論コンポーネント。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
DLR イメージ分類モデルストア	Greengrass アーティファクトとしてサンプル ResNet-50 イメージ分類モデルを含むモデルコンポーネント。	ジェネリック	Linux、Windows	いいえ
DLR オブジェクト検出モデルストア	Greengrass アーティファクトとしてサンプル YOLOv3 オブジェクト検出モデルを含むモデルコンポーネント。	ジェネリック	Linux、Windows	いいえ
DLR ランタイム	DLR とその従属関係を Greengrass コアデバイスにインストールするために使用されるインストールスクリプトを含むランタイムコンポーネント。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
TensorFlow Lite イメージ分類	TensorFlow Lite イメージ分類モデルストアと TensorFlow Lite ランタイムコンポーネントを依存関係として使用し、TensorFlow Lite をインストールし、サンプルイメージ分類モデルをダウンロードし、サポートされているデバイスでイメージ分類推論を実行する推論コンポーネント。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
TensorFlow ライトオブジェクト検出	TensorFlow Lite オブジェクト検出モデルストアと TensorFlow Lite ランタイムコンポーネントを依存関係として使用し、TensorFlow Lite をインストールし、サンプルオブジェクト検出モデルをダウンロードし、サポートされているデバイスでオブジェクト検出推論を実行する推論コンポーネント。	ジェネリック	Linux、Windows	いいえ
TensorFlow Lite イメージ分類モデルストア	Greengrass アーティファクトとしてサンプル MobileNet v1 モデルを含むモデルコンポーネント。	ジェネリック	Linux、Windows	いいえ

コンポーネント	説明	コンポーネントタイプ	サポートされる OS	オープンソース
TensorFlow Lite オブジェクト検出モデルストア	Greengrass アーティファクトとしてサンプルのシングルショット検出 (SSD) MobileNet モデルを含むモデルコンポーネント。	ジェネリック	Linux、Windows	いいえ
TensorFlow Lite ランタイム	Greengrass コアデバイスに TensorFlow Lite とその依存関係をインストールするために使用されるインストールスクリプトを含むランタイムコンポーネント。	ジェネリック	Linux、Windows	いいえ

Greengrass コアデバイスで Amazon SageMaker Edge Manager を使用する

> **Important**
>
> SageMaker Edge Manager は 2024 年 4 月 26 日に廃止されます。エッジデバイスにモデルをデプロイし続ける方法の詳細については、SageMaker 「Edge Manager のサポート終了」を参照してください。

Amazon SageMaker Edge Manager は、エッジデバイスで実行されるソフトウェアエージェントです。 SageMaker Edge Manager はエッジデバイスのモデル管理を提供するため、Amazon SageMaker Neo でコンパイルされたモデルを Greengrass コアデバイス上で直接パッケージ化して使用できます。 SageMaker Edge Manager を使用すると、コアデバイスからモデルの入力および出力データをサンプリングし、そのデータをモニタリングと分析AWS クラウドのために に送信することもできます。 SageMaker Edge Manager は SageMaker Neo を使用してターゲットハードウェアのモデルを最適化するため、DLR ランタイムをデバイスに直接インストールする必要はありません。Greengrass デバイスでは、 SageMaker Edge Manager はローカルAWS IoT証明書をロードしたり、AWS IoT認証情報プロバイダーエンドポイントを直接呼び出したりしません。代わりに、SageMaker Edge Manager はトークン交換サービスを使用して TES エンドポイントから一時的な認証情報を取得します。

このセクションでは、 SageMaker Edge Manager が Greengrass コアデバイスでどのように機能するかについて説明します。

SageMaker Edge Manager が Greengrass デバイスで動作する仕組み

SageMaker Edge Manager エージェントをコアデバイスにデプロイするには、`aws.greengrass.SageMakerEdgeManager`コンポーネントを含むデプロイを作成します。は、デバイス上の Edge Manager エージェントのインストールとライフサイクルAWS IoT Greengrassを管理します。エージェントバイナリの新しいバージョンが利用可能になったとき、`aws.greengrass.SageMakerEdgeManager` コンポーネントの更新されたバージョンをデプロイして、デバイスにインストールされているエージェントのバージョンをアップグレードします。

で SageMaker Edge Manager を使用する場合AWS IoT Greengrass、ワークフローには以下の大まかなステップが含まれます。

1. SageMaker Neo でモデルをコンパイルします。
2. SageMaker エッジパッケージングジョブを使用して SageMaker Neo コンパイル済みモデルをパッケージ化します。モデルにエッジパッケージングジョブを実行するとき、Greengrass コアデバイスにデプロイ可能なアーティファクトとして、パッケージ化されたモデルでモデルコンポーネントを作成することを選択できます。
3. カスタム推論コンポーネントを作成します。エッジマネージャーエージェントとやり取りしてコアデバイスで推論を実行するため、この推論コンポーネントを使用できます。これらの操作には、モデルのロード、推論を実行する予測リクエストの呼び出し、コンポーネントのシャットダウン時にモデルのアンロードが含まれます。

4. SageMaker Edge Manager コンポーネント、パッケージ化されたモデルコンポーネント、推論コンポーネントをデプロイして、デバイスの推論エンジン (Edge Manager エージェント) SageMaker でモデルを実行します。

SageMaker Edge Manager と連携するエッジパッケージングジョブと推論コンポーネントの作成の詳細については、Amazon SageMaker デベロッパーガイドの「 でモデルパッケージと Edge Manager エージェントをデプロイAWS IoT Greengrassする」を参照してください。

このチュートリアル: SageMaker Edge Manager の開始方法チュートリアルでは、サンプル推論およびモデルコンポーネントの作成に使用できる AWSが提供するサンプルコードを使用して、既存の Greengrass コアデバイスで SageMaker Edge Manager エージェントをセットアップして使用する方法を示します。

Greengrass コアデバイスで SageMaker Edge Manager を使用する場合、キャプチャデータ機能を使用してサンプルデータを にアップロードすることもできますAWS クラウド。キャプチャデータは、今後の分析のために S3 バケットまたはローカルディレクトリに推論入力、推論結果、および追加の推論データをアップロードするために使用する SageMaker 機能です。 SageMaker Edge Manager でキャプチャデータを使用する方法の詳細については、「Amazon SageMaker デベロッパーガイド」の「モデルの管理」を参照してください。

要件

Greengrass コアデバイスで SageMaker Edge Manager エージェントを使用するには、次の要件を満たす必要があります。

- Amazon Linux 2 で実行されている Greengrass コアデバイス、Debian ベースの Linux プラットフォーム (x86_64 またはArmv8)、または Windows (x86_64)。アカウントをお持ちでない場合は、「チュートリアル: AWS IoT Greengrass V2 の開始方法」を参照してください。
- Python 3.6 以降 (ご使用の Python のバージョン用 `pip` がコアデバイスにインストールされていること)。
- 次のように設定された Greengrass デバイスのロール:
  - 次の IAM ポリシーの例で示されているように、`credentials.iot.amazonaws.com` と `sagemaker.amazonaws.com` がロールの継承を可能にする信頼関係。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
```

```
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.iot.amazonaws.com"
       },
      "Action": "sts:AssumeRole"
    },
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "sagemaker.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

- IAM AmazonSageMakerEdgeDeviceFleetPolicy マネージドポリシー。
- 次の IAM ポリシーの例で示されている `s3:PutObject` アクション。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:PutObject"
      ],
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

- Greengrass コアデバイスAWS リージョンと同じ AWS アカウントと で作成された Amazon S3 バケット。 SageMaker Edge Manager では、エッジデバイスフリートを作成し、デバイスで実行中の推論からサンプルデータを保存するために S3 バケットが必要です。S3 バケットを作成する方法の情報については、「Amazon S3 の使用を開始」を参照してください。
- Greengrass コアデバイスと同じAWS IoTロールエイリアスを使用する SageMaker エッジデバイスフリート。詳細については、「エッジデバイスフリートを作成する」を参照してください。

- Greengrass コアデバイスは、 SageMaker エッジデバイスフリートにエッジデバイスとして登録されています。エッジデバイス名は、コアデバイスの AWS IoT モノ名に一致する必要があります。詳細については、「Greengrass コアデバイスを登録する」を参照してください。

SageMaker Edge Manager の使用を開始する

SageMaker Edge Manager の使用を開始するには、チュートリアルを完了します。このチュートリアルでは、既存のコアデバイスで AWSが提供するサンプルコンポーネントを使用して SageMaker Edge Manager の使用を開始する方法を示します。これらのサンプルコンポーネントは、 SageMaker Edge Manager コンポーネントを依存関係として使用して Edge Manager エージェントをデプロイし、 SageMaker Neo を使用してコンパイルされた事前トレーニング済みモデルを使用して推論を実行します。詳細については、「チュートリアル: SageMaker Edge Manager の開始方法」を参照してください。

Greengrass コアデバイスのAmazon Lookout for Vision を使用

> **Note**
>
> AWS IoT Greengrass は、現在 Windows コアデバイスにこの機能をサポートしていません。

Amazon Lookout for Vision は、産業製品の視覚的な欠陥を検出するために使用できる AWS のサービス です。産業製品の部品の欠落、車両や構造物の損傷、生産ラインの異常、プリント基板のコンデンサの欠落、シリコンウエハーや品質が重要視される他の物品の欠陥などに対し、コンピュータビジョンを使用して識別します。詳細については、「Amazon Lookout for Vision デベロッパーガイド」の「Amazon Lookout for Vision とは」を参照してください。

Lookout for Vision 推論を使用する Greengrass アプリケーションを作成して、Greengrass コアデバイスの視覚的な欠陥を検出できます。Lookout for Vision ワークフローを Greengrass コアデバイスにデプロイしたら、AWS クラウド の Lookout for Vision サービスに接続せずにコンピュータビジョンを実行できます。Lookout for Vision を使用する Greengrass アプリケーションを作成するには、次の Greengrass コンポーネントをセットアップしてデプロイします:

- Lookout for Vision モデルのコンポーネント - Greengrass アーティファクトとして Lookout for Vision 機械学習モデルが含まれています。Lookout for Vision コンソールと API を使用して、事前トレーニング済機械学習モデルをパッケージ化するモデルコンポーネントを生成できます。これらのコンポーネントは、AWS アカウント のプライベート Greengrass コンポーネントです。詳細に

ついては、「Amazon Lookout for Vision デベロッパーガイド」の「Lookout for Vision モデルの作成」と「Lookout for Vision モデルのパッケージ化」を参照してください。

- Lookout for Vision エッジエージェントコンポーネント - 指定した機械学習モデルを使用するコンピュータビジョンを使用して、異常を検出するローカルの Lookout for Vision ランタイムサーバーを提供します。このコンポーネントは、AWS が提供するコンポーネントです。詳細については、「Lookout for Vision エッジエージェントコンポーネント」を参照してください。
- Lookout for Vision クライアントアプリケーションコンポーネント - Lookout for Vision エッジエージェントコンポーネントと対話して、イメージを処理して異常を検出します。ローカルの Lookout for Vision エッジエージェントに画像やビデオストリームを送信し、機械学習モデルが検出する異常を報告するカスタムクライアントアプリケーションコンポーネントを開発できます。詳細については、「Amazon Lookout for Vision デベロッパーガイド」の「クライアントアプリケーションコンポーネントの記述」と「Lookout for Vision エッジエージェント API リファレンス」を参照してください。

これらのコンポーネントを作成、設定、使用する方法の詳細については、「Amazon Lookout for Vision デベロッパーガイド」の「エッジデバイスで Lookout for Vision モデルの使用」を参照してください。

機械学習コンポーネントのカスタマイズ

AWS IoT Greengrass では、推論、モデル、ランタイムコンポーネントを設定要素としてデバイスに機械学習の推論を実行する方法をカスタマイズするため、サンプルの機械学習コンポーネントを設定できます。AWS IoT Greengrass は、サンプルコンポーネントをテンプレートとして使用し、必要に応じて独自のカスタムコンポーネントを作成できる柔軟性も提供します。このモジュラーアプローチを組み合わせて、次の方法で機械学習の推論コンポーネントをカスタマイズできます。

サンプル推論コンポーネントの使用

- 推論コンポーネントをデプロイするときに設定を修正します。
- サンプルモデルストアコンポーネントをカスタムモデルコンポーネントに置き換えて、サンプル推論コンポーネントを備えたカスタムモデルを使用します。カスタムモデルは、サンプルモデルと同じランタイムを使用してトレーニングする必要があります。

カスタム推論コンポーネントの使用

- パブリックモデルコンポーネントとランタイムコンポーネントをカスタム推論コンポーネントの従属関係として追加して、サンプルモデルとランタイムを備えたカスタム推論コードを使用します。

- カスタムモデルコンポーネントまたはランタイムコンポーネントをカスタム推論コンポーネントの従属関係として作成して追加します。AWS IoT Greengrass がサンプルコンポーネントを提供しないカスタム推論コードまたはランタイムを使用する場合、カスタムコンポーネントを使用する必要があります。

トピック

パブリック推論コンポーネントの設定の修正

AWS IoT Greengrass コンソールで、コンポーネントページはコンポーネントのデフォルト設定を表示します。例えば、 TensorFlow Lite イメージ分類コンポーネントのデフォルト設定は次のようになります。

```
{
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
      "aws.greengrass.TensorFlowLiteImageClassification:mqttproxy:1": {
        "policyDescription": "Allows access to publish via topic ml/tflite/image-
classification.",
        "operations": [
          "aws.greengrass#PublishToIoTCore"
        ],
        "resources": [
          "ml/tflite/image-classification"
        ]
      }
    }
  },
  "PublishResultsOnTopic": "ml/tflite/image-classification",
  "ImageName": "cat.jpeg",
  "InferenceInterval": 3600,
  "ModelResourceKey": {
    "model": "TensorFlowLite-Mobilenet"
  }
```

```
}
```

パブリック推論コンポーネントをデプロイするとき、デフォルト設定を修正してデプロイをカスタマイズできます。各パブリック推論コンポーネントに利用可能な設定パラメータの情報については、AWS が提供する機械学習コンポーネント のコンポーネントトピックを参照してください。

このセクションでは、修正済コンポーネントを AWS IoT Greengrass コンソールからデプロイする方法について説明します。AWS CLI を使用してコンポーネントをデプロイする情報については、「デプロイの作成」を参照してください。

修正済パブリック推論コンポーネント (コンソール) をデプロイするには

1. AWS IoT Greengrass コンソール にサインインします。
2. ナビゲーションメニューで、[Components] (コンポーネント) を選択します。
3. [Components] (コンポーネント) ページの [Public components] (パブリックコンポーネント) タブで、デプロイするコンポーネントを選択します。
4. コンポーネントページで、[Deploy] (デプロイ) を選択します。
5. [Add to deployment] (デプロイに追加) から、次のいずれかを選択します。
   a. ターゲットデバイスにある既存のデプロイにこのコンポーネントをマージするには、[Add to existing deployment] (既存のデプロイに追加) をクリックし、修正するデプロイを選択します。
   b. ターゲットデバイスに新しいデプロイを作成するには、[Create new deployment] (新しいデプロイの作成) を選択します。デバイスに既存のデプロイがある場合は、このステップを選択すると既存のデプロイが置き換えられます。
6. [Specify device state] (ターゲットを指定) ページで、次を実行します。
   a. [Deployment information] (デプロイ情報) で、デプロイの名前を入力または変更して、わかりやすくします。
   b. [Deployment targets] (デプロイターゲット) でデプロイのターゲットを選択し、[Next] (次へ) を選択します。既存のデプロイを修正する場合は、デプロイターゲットを変更できません。
7. [Select components] (コンポーネントを選択) ページの [Public components] (パブリックコンポーネント) 内で、修正した設定を適用した推論コンポーネントが選択されていることを確認し、[Next] (次) を選択します。
8. [Configure components] (コンポーネントの設定) ページで、次の手順を実行します:

a. 推論コンポーネントを選択して、[Configure component] (コンポーネントの設定) を選択します。

b. [Configuration update] (設定更新) 内で、更新する設定値を入力します。例えば、[Configuration to merge] (マージする設定) ボックスに次の設定更新を入力し、推論間隔を 15 秒に変更して、`/custom-ml-inference/images/` フォルダに `custom.jpg` という名前のイメージを検索するようにコンポーネントに指示します。

```
{
  "InferenceInterval": "15",
  "ImageName": "custom.jpg",
  "ImageDirectory": "/custom-ml-inference/images/"
}
```

コンポーネントの設定を全体的にデフォルト値にリセットするには、[Reset paths] (パスのリセット) ボックスで空の文字列 "" を 1 つ指定します。

c. [Confirm] (確認)、[Next] (次へ) の順に選択します。

9. [Configure advanced setting] (高度な設定を設定) ページで、デフォルト構成設定のままにして [Next] (次) を選択します。

10. [Review] (レビュー) ページで、[Deploy] (デプロイ) を選択します。

サンプルの推論コンポーネントでカスタムモデルの使用

AWS IoT Greengrass がサンプルランタイムコンポーネントをランタイムに、サンプル推論コンポーネントを独自の機械学習モデルと使用する場合、これらのモデルをアーティファクトとして使用するコンポーネントで、パブリックモデルコンポーネントをオーバーライドする必要があります。高度なレベルで、次の手順を実行して、サンプル推論コンポーネントを備えたカスタムモデルを使用します。

1. S3 バケットのカスタムモデルをアーティファクトとして使用するモデルコンポーネントを作成します。カスタムモデルは、置き換えるモデルと同じランタイムを使用してトレーニングする必要があります。

2. カスタムモデルを使用するには、推論コンポーネントの `ModelResourceKey` 設定パラメータを修正します。推論コンポーネントの設定の更新に関する情報については、「パブリック推論コンポーネントの設定の修正」を参照してください。

推論コンポーネントをデプロイするとき、AWS IoT Greengrass はコンポーネント従属関係の最新バージョンを検索します。コンポーネントのそれ以降のカスタムバージョンが同じ AWS アカウントと AWS リージョン に存在する場合、依存するパブリックモデルコンポーネントをオーバーライドします。

カスタムモデルコンポーネント (コンソール) を作成

1. モデルを S3 バケットにアップロードします。S3 バケットにモデルをアップロードする情報については、「Amazon Simple Storage Service ユーザーガイド」の「Amazon S3 バケットの使用」を参照してください。

   > **Note**
   >
   > アーティファクトは、コンポーネントと同じ AWS アカウント と AWS リージョン の S3 バケットに格納する必要があります。AWS IoT Greengrass がこれらのアーティファクトにアクセスできるようにするには、Greengrass デバイスロールは `s3:GetObject` アクションを許可する必要があります。デバイスロールの詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

2. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Components] (コンポーネント) を選択します。
3. パブリックモデルのストアコンポーネントのコンポーネントレシピを取得します。

   a. [Components] (コンポーネント) ページの [Public components] (パブリックコンポーネント) タブで、新しいバージョンを作成するパブリックモデルコンポーネントを探して選択します。例えば、`variant.DLR.ImageClassification.ModelStore` です。

   b. コンポーネントページで、[View recipe] (レシピを確認) を選択して表示された JSON レシピをコピーします。

4. [Components] (コンポーネント) ページの [My components] (マイコンポーネント) タブで、[Create component] (コンポーネントの作成) を選択します。
5. [Create component] (コンポーネントの作成) ページの [Component information] (コンポーネント情報) 内で、[Enter recipe as JSON] (JSON としてレシピを入力) をコンポーネントソースとして選択します。
6. [Recipe] (レシピ) ボックスで、以前にコピーしたコンポーネントレシピを貼り付けます。
7. レシピで次の値を更新します。

   - `ComponentVersion`: コンポーネントのマイナーバージョンをインクリメントします。

カスタムコンポーネントを作成してパブリックモデルコンポーネントをオーバーライドする場合、既存のコンポーネントバージョンのマイナーバージョンのみを更新する必要があります。例えば、パブリックコンポーネントのバージョンが `2.1.0` の場合、バージョン `2.1.1` でカスタムコンポーネントを作成できます。

- `Manifests.Artifacts.Uri`: 各 URI 値を使用するモデルの Amazon S3 URI に更新します。

> **Note**
>
> コンポーネントの名前は変更しないでください。

8. [Create component] (コンポーネントの作成) を選択します。

カスタムモデルコンポーネントを作成 (AWS CLI)

1. モデルを S3 バケットにアップロードします。S3 バケットにモデルをアップロードする情報については、「Amazon Simple Storage Service ユーザーガイド」の「Amazon S3 バケットの使用」を参照してください。

> **Note**
>
> アーティファクトは、コンポーネントと同じ AWS アカウント と AWS リージョン の S3 バケットに格納する必要があります。AWS IoT Greengrass がこれらのアーティファクトにアクセスできるようにするには、Greengrass デバイスロールは `s3:GetObject` アクションを許可する必要があります。デバイスロールの詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

2. 次のコマンドを実行して、パブリックコンポーネントのコンポーネントレシピを取得します。このコマンドは、コマンドで指定した出力ファイルにコンポーネントレシピを書き込みます。必要に応じて、取得した Base64 でエンコードされた文字列を JSON または YAML に変換します。

   Linux, macOS, or Unix

```
aws greengrassv2 get-component \
    --arn <arn> \
    --recipe-output-format <recipe-format> \
    --query recipe \
```

```
    --output text | base64 --decode > <recipe-file>
```

Windows Command Prompt (CMD)

```
aws greengrassv2 get-component ^
    --arn <arn> ^
    --recipe-output-format <recipe-format> ^
    --query recipe ^
    --output text > <recipe-file>.base64

certutil -decode <recipe-file>.base64 <recipe-file>
```

PowerShell

```
aws greengrassv2 get-component `
    --arn <arn> `
    --recipe-output-format <recipe-format> `
    --query recipe `
    --output text > <recipe-file>.base64

certutil -decode <recipe-file>.base64 <recipe-file>
```

3. レシピファイルの名前を *<component-name>*-*<component-version>* に更新します。ここでは、コンポーネントバージョンは新しいコンポーネントのターゲットバージョンです。例えば、`variant.DLR.ImageClassification.ModelStore-2.1.1.yaml` です。
4. レシピで次の値を更新します。

   - `ComponentVersion`: コンポーネントのマイナーバージョンをインクリメントします。

     カスタムコンポーネントを作成してパブリックモデルコンポーネントをオーバーライドする場合、既存のコンポーネントバージョンのマイナーバージョンのみを更新する必要があります。例えば、パブリックコンポーネントのバージョンが `2.1.0` の場合、バージョン `2.1.1` でカスタムコンポーネントを作成できます。

   - `Manifests.Artifacts.Uri`: 各 URI 値を使用するモデルの Amazon S3 URI に更新します。

> **Note**
>
> コンポーネントの名前は変更しないでください。

5. 次のコマンドを実行して、取得して修正したレシピを使用して新しいコンポーネントを作成します。

```
aws greengrassv2 create-component-version \
    --inline-recipe fileb://path/to/component/recipe
```

> **Note**
>
> この手順では、AWS クラウド の AWS IoT Greengrass サービスのコンポーネントを作成します。コンポーネントをクラウドにアップロードする前に、Greengrass CLI を使用してコンポーネントをローカルで開発、テスト、デプロイできます。詳細については、「AWS IoT Greengrass コンポーネントを開発する」を参照してください。

コンポーネントの作成に関する詳細については、「AWS IoT Greengrass コンポーネントを開発する」を参照してください。

カスタム機械学習のコンポーネントを作成

AWS IoT Greengrass がサンプルコンポーネントを提供しないカスタム推論コードまたはランタイムを使用する場合、カスタムコンポーネントを作成する必要があります。カスタム推論コードを AWS が提供するサンプル機械学習モデルとランタイムと使用することができます。または、独自のモデルとランタイムを使用した完全にカスタマイズされた機械学習の推論ソリューションを開発できます。モデルが、AWS IoT Greengrass がサンプルランタイムコンポーネントを提供するランタイムを使用している場合、そのランタイムコンポーネントを使用できますが、推論コードと使用するモデルに対してのみカスタムコンポーネントを作成する必要があります。

トピック

- パブリックコンポーネントのレシピを取得
- サンプルコンポーネントのアーティファクトを取得
- S3 バケットへコンポーネントアーティファクトのアップロード
- カスタムコンポーネントの作成

パブリックコンポーネントのレシピを取得

既存のパブリック機械学習コンポーネントのレシピをテンプレートとして使用して、カスタムコンポーネントを作成できます。パブリックコンポーネントの最新バージョン用のコンポーネントレシピを確認するには、コンソールまたは AWS CLI を次のように使用します。

- コンソールの使用

  1. [Components] (コンポーネント) ページの [Public components] (パブリックコンポーネント) タブで、パブリックコンポーネントを探して選択します。
  2. コンポーネントページで、[View recipe] (レシピを確認) を選択します。

- AWS CLI を使用する

  次のコマンドを実行して、パブリックバリアントコンポーネントのコンポーネントレシピを取得します。このコマンドは、コマンドで指定した JSON または YAML レシピファイルにコンポーネントレシピを書き込みます。

  Linux, macOS, or Unix

  ```
  aws greengrassv2 get-component \
      --arn <arn> \
      --recipe-output-format <recipe-format> \
      --query recipe \
      --output text | base64 --decode > <recipe-file>
  ```

  Windows Command Prompt (CMD)

  ```
  aws greengrassv2 get-component ^
      --arn <arn> ^
      --recipe-output-format <recipe-format> ^
      --query recipe ^
      --output text > <recipe-file>.base64

  certutil -decode <recipe-file>.base64 <recipe-file>
  ```

  PowerShell

  ```
  aws greengrassv2 get-component `
      --arn <arn> `
      --recipe-output-format <recipe-format> `
      --query recipe `
  ```

```
    --output text > <recipe-file>.base64

certutil -decode <recipe-file>.base64 <recipe-file>
```

コマンドの値を、次のように置き換えます。

- *<arn>*: パブリックコンポーネントの Amazon リソースネーム (ARN)。
- *<recipe-format>*: レシピファイルを作成するフォーマット。サポートされている値は、`JSON` および `YAML` です。
- *<recipe-file>*: *<component-name>*-*<component-version>* フォーマットのレシピの名前。

サンプルコンポーネントのアーティファクトを取得

パブリック機械学習コンポーネントで使用されるアーティファクトをテンプレートとして使用して、推論コードまたはランタイムインストールスクリプトなど、カスタムコンポーネントのアーティファクトを作成できます。

パブリック機械学習コンポーネントに含まれるサンプルアーティファクトを確認するには、パブリック推論コンポーネントをデプロイし、*/greengrass/v2*`/packages/artifacts-unarchived/`*component-name*`/`*component-version*`/` フォルダのデバイスのアーティファクトを確認します。

S3 バケットへコンポーネントアーティファクトのアップロード

カスタムコンポーネントを作成する前に、コンポーネントアーティファクトを S3 バケットにアップロードし、コンポーネントレシピの S3 URI を使用する必要があります。例えば、推論コンポーネントでカスタム推論コードを使用するには、S3 バケットにコードをアップロードします。次に、推論コードの Amazon S3 URI をコンポーネントのアーティファクトとして使用できます。

S3 バケットにコンテンツのアップロードに関する情報については、「Amazon Simple Storage Service ユーザーガイド」の「Amazon S3 バケットの操作」を参照してください。

> **Note**
>
> アーティファクトは、コンポーネントと同じ AWS アカウント と AWS リージョン の S3 バケットに格納する必要があります。AWS IoT Greengrass がこれらのアーティファクトにアクセスできるようにするには、Greengrass デバイスロールは `s3:GetObject` アクションを

許可する必要があります。デバイスロールの詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

カスタムコンポーネントの作成

取得したアーティファクトとレシピを使用してカスタム機械学習コンポーネントを作成できます。例については「カスタム推論コンポーネントを作成」を参照してください。

コンポーネントの作成と Greengrass デバイスへのデプロイの詳細については、「AWS IoT Greengrass コンポーネントを開発する」と「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

カスタム推論コンポーネントを作成

このセクションでは、DLR イメージ分類コンポーネントをテンプレートとして使用して、カスタム推論コンポーネントを作成する方法を説明します。

トピック

- 推論コードを Amazon S3 バケットにアップロード
- 推論コンポーネントのレシピを作成
- 推論コンポーネントの作成

推論コードを Amazon S3 バケットにアップロード

推論コードを作成し、S3 バケットにアップロードします。S3 バケットにコンテンツのアップロードに関する情報については、「Amazon Simple Storage Service ユーザーガイド」の「Amazon S3 バケットの操作」を参照してください。

> **Note**
>
> アーティファクトは、コンポーネントと同じ AWS アカウント と AWS リージョン の S3 バケットに格納する必要があります。AWS IoT Greengrass がこれらのアーティファクトにアクセスできるようにするには、Greengrass デバイスロールは `s3:GetObject` アクションを許可する必要があります。デバイスロールの詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

推論コンポーネントのレシピを作成

1. 次のコマンドを実行して、DLR イメージ分類コンポーネントのコンポーネントレシピを取得します。このコマンドは、コマンドで指定した JSON または YAML レシピファイルにコンポーネントレシピを書き込みます。

   Linux, macOS, or Unix

   ```
   aws greengrassv2 get-component \
       --arn
    arn:aws:greengrass:region:aws:components:aws.greengrass.DLRImageClassification:versions
    \
       --recipe-output-format JSON | YAML \
       --query recipe \
       --output text | base64 --decode > <recipe-file>
   ```

   Windows Command Prompt (CMD)

   ```
   aws greengrassv2 get-component ^
       --arn
    arn:aws:greengrass:region:aws:components:aws.greengrass.DLRImageClassification:versions
    ^
       --recipe-output-format JSON | YAML ^
       --query recipe ^
       --output text > <recipe-file>.base64

   certutil -decode <recipe-file>.base64 <recipe-file>
   ```

   PowerShell

   ```
   aws greengrassv2 get-component `
       --arn
    arn:aws:greengrass:region:aws:components:aws.greengrass.DLRImageClassification:versions
    `
       --recipe-output-format JSON | YAML `
       --query recipe `
       --output text > <recipe-file>.base64

   certutil -decode <recipe-file>.base64 <recipe-file>
   ```

<recipe-file> を *<component-name>*-*<component-version>* 形式のレシピの名前に置き換えます。

2. レシピの ComponentDependencies オブジェクトに、使用するモデルとランタイムコンポーネントに応じて次の操作を 1 つ以上実行します。

   - DLR コンパイル型モデルを使用する場合、DLR コンポーネントの従属関係を維持します。次の例で示すように、カスタム ランタイム コンポーネントの従属関係に置き換えることもできます。

     ランタイムコンポーネント

     JSON

     ```
     {
         "<runtime-component>": {
             "VersionRequirement": "<version>",
             "DependencyType": "HARD"
         }
     }
     ```

     YAML

     ```
     <runtime-component>:
         VersionRequirement: "<version>"
         DependencyType: HARD
     ```

   - DLR イメージ分類モデルストアの依存関係を維持して、 AWSが提供する事前トレーニング済みの ResNet-50 モデルを使用するか、カスタムモデルコンポーネントを使用するように変更します。パブリックモデルコンポーネントの従属関係を含めるとき、そのコンポーネントの新しいカスタムバージョンが同じ AWS アカウント と AWS リージョン に存在する場合、推論コンポーネントはそのカスタムコンポーネントを使用します。次の例に示すように、モデルコンポーネントの従属関係を指定します。

     パブリックモデルコンポーネント

     JSON

     ```
     {
         "variant.DLR.ImageClassification.ModelStore": {
             "VersionRequirement": "<version>",
     ```

```
        "DependencyType": "HARD"
    }
}
```

YAML

```
variant.DLR.ImageClassification.ModelStore:
    VersionRequirement: "<version>"
    DependencyType: HARD
```

カスタムモデルコンポーネント

JSON

```
{
    "<custom-model-component>": {
        "VersionRequirement": "<version>",
        "DependencyType": "HARD"
    }
}
```

YAML

```
<custom-model-component>:
    VersionRequirement: "<version>"
    DependencyType: HARD
```

3. ComponentConfiguration オブジェクトで、このコンポーネントのデフォルト設定を追加します。コンポーネントをデプロイするときに、後でこの設定を修正できます。次の抜粋は、DLR イメージ分類コンポーネントのコンポーネント設定を示しています。

   例えば、カスタム推論コンポーネントの従属関係としてカスタムモデルコンポーネントを使用する場合、使用しているモデルの名前を提供するように ModelResourceKey を修正します。

   JSON

```
{
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
      "aws.greengrass.ImageClassification:mqttproxy:1": {
        "policyDescription": "Allows access to publish via topic ml/dlr/image-
classification.",
```

```
        "operations": [
          "aws.greengrass#PublishToIoTCore"
        ],
        "resources": [
          "ml/dlr/image-classification"
        ]
      }
    }
  },
  "PublishResultsOnTopic": "ml/dlr/image-classification",
  "ImageName": "cat.jpeg",
  "InferenceInterval": 3600,
  "ModelResourceKey": {
    "armv7l": "DLR-resnet50-armv7l-cpu-ImageClassification",
    "x86_64": "DLR-resnet50-x86_64-cpu-ImageClassification",
    "aarch64": "DLR-resnet50-aarch64-cpu-ImageClassification"
  }
}
```

YAML

```
accessControl:
    aws.greengrass.ipc.mqttproxy:
        'aws.greengrass.ImageClassification:mqttproxy:1':
            policyDescription: 'Allows access to publish via topic ml/dlr/image-
classification.'
            operations:
                - 'aws.greengrass#PublishToIoTCore'
            resources:
                - ml/dlr/image-classification
PublishResultsOnTopic: ml/dlr/image-classification
ImageName: cat.jpeg
InferenceInterval: 3600
ModelResourceKey:
    armv7l: "DLR-resnet50-armv7l-cpu-ImageClassification"
    x86_64: "DLR-resnet50-x86_64-cpu-ImageClassification"
    aarch64: "DLR-resnet50-aarch64-cpu-ImageClassification"
```

4. Manifests オブジェクトで、コンポーネントが異なるプラットフォームにデプロイされるときに使用されるこのコンポーネントのアーティファクトと設定に関する情報、並びにコンポーネントを正常に実行するために必要な他の情報を提供します。次の抜粋では、DLR イメージ分類コ

ンポーネントの Linux プラットフォームの Manifests オブジェクトに対する設定を示しています。

JSON

```
{
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "architecture": "arm"
      },
      "Name": "32-bit armv7l - Linux (raspberry pi)",
      "Artifacts": [
        {
          "URI": "s3://SAMPLE-BUCKET/sample-artifacts-directory/
image_classification.zip",
          "Unarchive": "ZIP"
        }
      ],
      "Lifecycle": {
        "Setenv": {
          "DLR_IC_MODEL_DIR":
 "{variant.DLR.ImageClassification.ModelStore:artifacts:decompressedPath}/
{configuration:/ModelResourceKey/armv7l}",
          "DEFAULT_DLR_IC_IMAGE_DIR": "{artifacts:decompressedPath}/
image_classification/sample_images/"
        },
        "run": {
          "RequiresPrivilege": true,
          "script": ". {variant.DLR:configuration:/MLRootPath}/
greengrass_ml_dlr_venv/bin/activate\npython3 {artifacts:decompressedPath}/
image_classification/inference.py"
        }
      }
    }
  ]
}
```

YAML

```
Manifests:
  - Platform:
```

```
      os: linux
      architecture: arm
    Name: 32-bit armv7l - Linux (raspberry pi)
    Artifacts:
      - URI: s3://SAMPLE-BUCKET/sample-artifacts-directory/
image_classification.zip
        Unarchive: ZIP
    Lifecycle:
      Setenv:
        DLR_IC_MODEL_DIR:
 "{variant.DLR.ImageClassification.ModelStore:artifacts:decompressedPath}/
{configuration:/ModelResourceKey/armv7l}"
        DEFAULT_DLR_IC_IMAGE_DIR: "{artifacts:decompressedPath}/
image_classification/sample_images/"
      run:
        RequiresPrivilege: true
        script: |-
          . {variant.DLR:configuration:/MLRootPath}/greengrass_ml_dlr_venv/bin/
activate
          python3 {artifacts:decompressedPath}/image_classification/inference.py
```

コンポーネントレシピの作成の詳細については、「AWS IoT Greengrass コンポーネントレシピのリファレンス」を参照してください。

推論コンポーネントの作成

AWS IoT Greengrass コンソールまたは AWS CLI を使用して、今定義したレシピを使用してコンポーネントを作成します。コンポーネントを作成した後、デバイスに推論を実行するためにデプロイできます。推論コンポーネントのデプロイ方法の例については、「チュートリアル: TensorFlow Lite を使用してサンプルイメージ分類推論を実行する」を参照してください。

カスタム推論コンポーネント (コンソール) を作成

1. AWS IoT Greengrass コンソール にサインインします。
2. ナビゲーションメニューで、[Components] (コンポーネント) を選択します。
3. [Components] (コンポーネント) ページの [My components] (マイコンポーネント) タブで、[Create component] (コンポーネントの作成) を選択します。
4. [Create component] (コンポーネントの作成) ページの [Component information] (コンポーネント情報) 内で、[Enter recipe as JSON] (JSON としてレシピを入力) または [Enter recipe as YAML] (YAML としてレシピを入力) をコンポーネントソースとして選択します。

5. [Recipe] (レシピ) ボックスで、作成したカスタムレシピを入力します。
6. [Create component] (コンポーネントの作成) をクリックします。

カスタム推論コンポーネント (AWS CLI) を作成

次のコマンドを実行し、作成したレシピを使用して、新しいカスタムコンポーネントを作成します。

```
aws greengrassv2 create-component-version \
    --inline-recipe fileb://path/to/recipe/file
```

> **Note**
>
> この手順では、AWS クラウド の AWS IoT Greengrass サービスのコンポーネントを作成します。コンポーネントをクラウドにアップロードする前に、Greengrass CLI を使用してコンポーネントをローカルで開発、テスト、デプロイできます。詳細については、「AWS IoT Greengrass コンポーネントを開発する」を参照してください。

機械学習の推論に対するトラブルシューティング

このセクションのトラブルシューティング情報と解決策を使用して、機械学習コンポーネントの問題を解決してください。パブリック機械学習の推論コンポーネントについては、次のコンポーネントログのエラーメッセージを参照してください。

Linux or Unix

- */greengrass/v2*/logs/aws.greengrass.DLRImageClassification.log
- */greengrass/v2*/logs/aws.greengrass.DLRObjectDetection.log
- */greengrass/v2*/logs/aws.greengrass.TensorFlowLiteImageClassification.log
- */greengrass/v2*/logs/aws.greengrass.TensorFlowLiteObjectDetection.log

Windows

- *C:\greengrass\v2*\logs\aws.greengrass.DLRImageClassification.log
- *C:\greengrass\v2*\logs\aws.greengrass.DLRObjectDetection.log

- *C:\greengrass\v2*\logs\aws.greengrass.TensorFlowLiteImageClassification.log
- *C:\greengrass\v2*\logs\aws.greengrass.TensorFlowLiteObjectDetection.log

コンポーネントが正しくインストールされている場合、コンポーネントログに推論に使用するライブラリの場所が含まれます。

問題

ライブラリのフェッチに失敗しました

次のエラーは、Raspberry Pi デバイスへのデプロイ中にインストーラスクリプトが必要なライブラリのダウンロードに失敗したときに発生します。

```
Err:2 http://raspbian.raspberrypi.org/raspbian buster/main armhf python3.7-dev armhf
 3.7.3-2+deb10u1
404 Not Found [IP: 93.93.128.193 80]
E: Failed to fetch http://raspbian.raspberrypi.org/raspbian/pool/main/p/python3.7/
libpython3.7-dev_3.7.3-2+deb10u1_armhf.deb 404 Not Found [IP: 93.93.128.193 80]
```

`sudo apt-get update` を実行してコンポーネントを再度デプロイします。

Cannot open shared object file

インストーラスクリプトが、Raspberry Pi デバイスへのデプロイ時に `opencv-python` の必要な従属関係のダウンロードに失敗するとき、次のようなエラーが表示される場合があります。

```
ImportError: libopenjp2.so.7: cannot open shared object file: No such file or directory
```

以下のコマンドを実行して、opencv-pythonの従属関係を手動でインストールします:

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
 libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

Error: ModuleNotFoundError: No module named '<library>'

ML ランタイムライブラリまたはその依存関係が正しくインストールされていないとき、ML ランタイム コンポーネント ログ (`variant.DLR.log` または `variant.TensorFlowLite.log`) にこのエラーが表示されることがあります。このエラーは、次の状況に発生する可能性があります:

- デフォルトで有効になっている `UseInstaller` オプションを使用する場合、このエラーは、ML ランタイムコンポーネントがランタイムまたはその従属関係のインストールに失敗したことを示します。以下の操作を実行します。
  1. ML ランタイムコンポーネントを設定して `UseInstaller` オプションを無効にすします。
  2. ML ランタイムとその従属関係をインストールし、ML コンポーネントを実行するシステムユーザーがそれらを利用できるようにします。詳細については、次を参照してください。
     - DLR ランタイム UseInstaller オプション
     - TensorFlow Lite ランタイム UseInstaller オプション
- `UseInstaller` オプションを使用しない場合、このエラーは、ML コンポーネントを実行するシステムユーザーに ML ランタイムまたはその従属関係がインストールされていないことを示します。以下の操作を実行します。
  1. ML コンポーネントを実行するシステムユーザー用にライブラリがインストールされていることを確認します。*`ggc_user`* をシステムユーザーの名前に置き換えて、*`tflite_runtime`* を確認するライブラリの名前に置き換えます。

     Linux or Unix

     ```
     sudo -H -u ggc_user bash -c "python3 -c 'import tflite_runtime'"
     ```

     Windows

     ```
     runas /user:ggc_user "py -3 -c \"import tflite_runtime\""
     ```

2. ライブラリがインストールされていない場合は、そのユーザー用にインストールします。*ggc_user* をシステムユーザーの名前に置き換えて、*tflite_runtime* をライブラリの名前に置き換えます。

   Linux or Unix

   ```
   sudo -H -u ggc_user bash -c "python3 -m pip install --user tflite_runtime"
   ```

   Windows

   ```
   runas /user:ggc_user "py -3 -m pip install --user tflite_runtime"
   ```

   各 ML ランタイムの従属関係の詳細については、次を参照してください。

   - DLR ランタイム UseInstaller オプション
   - TensorFlow Lite ランタイム UseInstaller オプション

3. 問題が解決しない場合、別のユーザー用にライブラリをインストールして、このデバイスでライブラリをインストールできるかどうかを確認します。例えば、ユーザーは、お客様のユーザー、ルートユーザー、管理者ユーザーの場合があります。どのユーザーに対してもライブラリを正常にインストールできない場合、デバイスがライブラリをサポートしていない可能性があります。ライブラリのマニュアルを参照して、要件を確認してインストールの問題のトラブルシューティングを行ってください。

CUDA 対応デバイスが検出されません

GPU アクセラレーションを使用すると、次のエラーが表示されることがあります。次のコマンドを実行して、Greengrass ユーザーの GPU アクセスを有効にします。

```
sudo usermod -a -G video ggc_user
```

そのようなファイルまたはディレクトリはありません

次のエラーは、ランタイムコンポーネントが仮想環境を正しく設定できなかったことを示します:

- *MLRootPath*/greengrass_ml_dlr_conda/bin/conda: No such file or directory
- *MLRootPath*/greengrass_ml_dlr_venv/bin/activate: No such file or directory
- *MLRootPath*/greengrass_ml_tflite_conda/bin/conda: No such file or directory

- *MLRootPath*/greengrass_ml_tflite_venv/bin/activate: No such file or directory

ログをチェックして、すべてのランタイム従属関係が正しくインストールされていることを確認します。インストーラスクリプトによってインストールされたライブラリの詳細については、次のトピックを参照してください。

- DLR ランタイム
- TensorFlow Lite ランタイム

デフォルトでは、*MLRootPath* は に設定されています*/greengrass/v2*/work/*component-name*/greengrass_ml。この場所を変更するには、DLR ランタイム または TensorFlow Lite ランタイム ランタイムコンポーネントをデプロイに直接含めて、MLRootPath パラメータの修正値を設定マージ更新に指定です。コンポーネントの設定の詳細については、「コンポーネント設定の更新」を参照してください。

> **Note**
>
> DLR コンポーネント v1.3.x の場合、推論コンポーネントの設定で MLRootPath パラメータを設定し、デフォルト値は $HOME/greengrass_ml です。

RuntimeError: module compiled against API version 0xf but this version of NumPy is <version>

Raspberry Pi OS Bullseye を実行している Raspberry Pi で機械学習の推論を実行するとき、次のエラーが表示される場合があります。

```
RuntimeError: module compiled against API version 0xf but this version of numpy is 0xd
ImportError: numpy.core.multiarray failed to import
```

このエラーは、Raspberry Pi OSwireseye に OpenCV が必要とするバージョン NumPyより前のバージョンの が含まれているために発生します。この問題を解決するには、次のコマンドを実行して最新バージョン NumPy にアップグレードします。

```
pip3 install --upgrade numpy
```

picamera.exc.PiCameraError: Camera is not enabled

Raspberry Pi OS Bullseye を実行している Raspberry Pi で機械学習の推論を実行するとき、次のエラーが表示される場合があります。

```
picamera.exc.PiCameraError: Camera is not enabled. Try running 'sudo raspi-config' and
 ensure that the camera has been enabled.
```

このエラーは、Raspberry Pi OS Bullseye に ML コンポーネントと互換性のない新しいカメラスタックが含まれているために発生します。この問題を解決するには、レガシーカメラスタックを有効にします。

レガシーカメラスタックを有効にするには

1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

   ```
   sudo raspi-config
   ```

2. [Interface Options] (インターフェイスオプション) を選択します。
3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
4. Raspberry Pi を再起動します。

メモリエラー

次のエラーは、通常、デバイスに十分なメモリがなく、コンポーネントプロセスが中断された場合に発生します。

- `stderr. Killed.`
- `exitCode=137`

パブリック機械学習の推論コンポーネントをデプロイするには、最低 500 MB のメモリを推奨します。

ディスク容量エラー

`no space left on device` エラーは、通常、デバイスに十分なストレージがない場合に発生します。コンポーネントを再度デプロイする前に、デバイスに十分なディスク容量があることを確認

してください。パブリック機械学習の推論コンポーネントをデプロイするには、最低 500 MB の空きディスク容量をお勧めします。

タイムアウトエラー

パブリック機械学習コンポーネントは、200 MB を超える大きな機械学習モデルファイルをダウンロードします。デプロイ中にダウンロードがタイムアウトした場合、インターネット接続速度を確認してデプロイを再試行してください。

AWS Systems Manager で Greengrass コアデバイスを管理する

> **Note**
>
> AWS IoT Greengrass は、現在 Windows コアデバイスにこの機能をサポートしていません。

Systems Manager は、Amazon EC2 インスタンス、オンプレミスのサーバーや仮想マシン (VM)、エッジデバイスなど、AWS のインフラストラクチャを表示および制御するために使用できる AWS サービスです。Systems Manager は、運用データの表示、運用タスクの自動化、セキュリティとコンプライアンスの維持を可能にします。Systems Manager でマシンを登録するとき、マネージドノードと呼ばれます。詳細については、『AWS Systems Manager ユーザーガイド』の「What is AWS Systems Manager? (とは ?)」 を参照してください。

AWS Systems Manager Agent (Systems Manager Agent) は、Systems Manager がデバイスを更新、管理、設定できるため、デバイスにインストールできるソフトウェアです。Greengrass コアデバイスに Systems Manager Agent をインストールするには、Systems Manager Agent コンポーネントをデプロイします。Systems Manager Agent を初めてデプロイするとき、コアデバイスを Systems Manager マネージドノードとして登録します。Systems Manager Agent はデバイスに実行されて、AWS クラウド の Systems Manager サービスとの通信を有効にします。Systems Manager Agent コンポーネントのインストールと設定方法の詳細については、「AWS Systems Manager エージェントのインストール」を参照してください。

Systems Manager のツールや特徴は、機能と呼ばれます。Greengrass コアデバイスは、すべての Systems Manager 機能をサポートしています。これらの機能と Systems Manager を使用してコアデバイスを管理する方法の詳細については、「AWS Systems Manager ユーザーガイド」の「Systems Manager 機能」を参照してください。

AWS Systems Manager は、Systems Manager マネージドノードに標準インスタンス層と高度インスタンス層を提供します。Systems Manager を初めて使用する場合、標準インスタンス層で開始します。標準インスタンス層では、AWS アカウント で AWS リージョン ごとに最大 1,000 個のマネージドノードを登録できます。1 つのアカウントとリージョンに 1,000 個を超えるマネージドノードを登録する必要がある場合、あるいは Session Manager 機能を使用する必要がある場合、高度インスタンス層を使用してください。詳細については、「AWS Systems Manager ユーザーガイド」の「インスタンス層の設定」を参照してください。

トピック

AWS Systems Manager エージェントのインストール

AWS Systems Manager エージェント (Systems Manager Agent) は Amazon のソフトウェアで、インストールすることにより Greengrass コアデバイス、Amazon EC2 インスタンス、その他のリソースを Systems Manager が更新、管理、設定できるようになります。エージェントは、AWS クラウド の Systems Manager サービスからのリクエストを処理および実行します。次に、エージェントは Systems Manager サービスにステータスとランタイム情報を返します。詳細については、「AWS Systems Manager ユーザーガイド」の「Systems Manager Agent について」を参照してください。

AWS は、Systems Manager Agent を Greengrass コンポーネントとして提供します。これは、Greengrass コアデバイスにデプロイして Systems Manager で管理できるようにするものです。Systems Manager Agent コンポーネントは、Systems Manager Agent ソフトウェアをインストールし、コアデバイスをマネージドノードとして Systems Manager に登録します。このページの手順に従って前提条件を完了し、Systems Manager Agent コンポーネントをコアデバイスまたはコアデバイスのグループにデプロイします。

トピック

ステップ 1: Systems Manager の一般的なセットアップ手順を完了する

まだ実行していない場合、AWS Systems Manager の一般的なセットアップ手順を完了してください。詳細については、「AWS Systems Manager ユーザーガイド」の「Systems Manager の一般的なセットアップ手順を完了する」を参照してください。

ステップ 2: Systems Manager の IAM サービスロールを作成する

Systems Manager Agent は AWS Identity and Access Management (IAM) サービスロール を使用して AWS Systems Manager と通信します。Systems Manager は、各コアデバイスで Systems Manager の機能を有効にするために、このロールを引き受けます。また、コンポーネントのデプロイ時にコアデバイスを Systems Manager マネージドノードとして登録する際にも、Systems Manager Agent コンポーネントはこのロールを使用します。まだ作成していない場合は、Systems Manager Agent コンポーネントが使用する Systems Manager サービスロールを作成します。詳細については、「AWS Systems Manager ユーザーガイド」の「エッジデバイス用の IAM サービスロールを作成」を参照してください。

ステップ 3: アクセス権限をトークン交換ロールに追加する

Greengrass コアデバイスは、トークン交換ロールという IAM サービスロールを使用して、AWS サービスとやり取りを行います。各コアデバイスには、AWS IoT Greengrass Core ソフトウェアをインストールする際に作成するトークン交換ロールがあります。Systems Manager Agent などの多くの Greengrass コンポーネントには、このロールに対する追加のアクセス権限が必要です。Systems Manager Agent コンポーネントには、ステップ 2: Systems Manager の IAM サービスロールを作成する で作成したロールを使用するアクセス権限を含む、以下のアクセス権限が必要です。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "iam:PassRole"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:iam::account-id:role/SSMServiceRole"
      ]
    },
    {
      "Action": [
        "ssm:AddTagsToResource",
        "ssm:RegisterManagedInstance"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
```

```
  ]
}
```

まだ追加していない場合は、これらのアクセス権限をコアデバイスのトークン交換ロールに追加して、Systems Manager Agent が動作できるようにします。トークン交換ロールに新しいポリシーを追加することで、このアクセス許可を付与できます。

アクセス権限をトークン交換ロールに追加するには (コンソール)

1. [IAM console] (IAM コンソール) ナビゲーションメニューで、[Roles] (ロール) を選択します。
2. AWS IoT Greengrass Core ソフトウェアをインストールする際にトークン交換ロールとして設定した IAM ロールを選択します。AWS IoT Greengrass Core ソフトウェアのインストール時にトークン交換ロールの名前を指定しなかった場合は、`GreengrassV2TokenExchangeRole` という名前のロールが作成されます。
3. [Permissions] (アクセス許可) タブを選択し、[Add permissions] (アクセス許可の追加) を選択してから、[Attach policies] (ポリシーの添付) を選択します。
4. [Create policy] (ポリシーの作成) を選択します。[ポリシーの作成] ページが新しいブラウザタブで開きます。
5. [ポリシーの作成] ページで、次の操作を行います。

   a. [JSON] を選択して JSON エディタを開きます。

   b. 以下の ポリシーを JSON エディタに貼り付けます。*`SSMServiceRole`* を で作成したサービスロールの名前に置き換えますステップ 2: Systems Manager の IAM サービスロールを作成する。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "iam:PassRole"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:iam::account-id:role/SSMServiceRole"
      ]
    },
    {
      "Action": [
```

```
        "ssm:AddTagsToResource",
        "ssm:RegisterManagedInstance"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

c. [次へ: タグ] を選択します。

d. [次へ: レビュー] を選択します。

e. ポリシーの [名前] を入力します (例: **GreengrassSSMAgentComponentPolicy**)。

f. [ポリシーの作成] を選択します。

g. 先にトークン交換ロールを開いていたブラウザタブに切り替えます。

6. [Add permissions] (アクセス許可を追加) ページで更新ボタンを選択して、前のステップで作成した Greengrass Systems Manager エージェントポリシーを選択します。

7. [ポリシーのアタッチ] を選択します。

   これで、このトークン交換ロールを使用するコアデバイスには、Systems Manager サービスとやり取りする権限が付与されます。

アクセス権限をトークン交換ロールに追加するには (AWS CLI)

Systems Manager を使用するアクセス許可を付与するポリシーを追加するには

1. `ssm-agent-component-policy.json` という名前のファイルを作成して、次の JSON をファイルにコピーします。*`SSMServiceRole`* を で作成したサービスロールの名前に置き換えますステップ 2: Systems Manager の IAM サービスロールを作成する。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "iam:PassRole"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:iam::account-id:role/SSMServiceRole"
```

```
      ]
    },
    {
      "Action": [
        "ssm:AddTagsToResource",
        "ssm:RegisterManagedInstance"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

2. 次のコマンドを実行して、ssm-agent-component-policy.json のポリシードキュメントからポリシーを作成します。

   Linux or Unix

   ```
   aws iam create-policy \
     --policy-name GreengrassSSMAgentComponentPolicy \
     --policy-document file://ssm-agent-component-policy.json
   ```

   Windows Command Prompt (CMD)

   ```
   aws iam create-policy ^
     --policy-name GreengrassSSMAgentComponentPolicy ^
     --policy-document file://ssm-agent-component-policy.json
   ```

   PowerShell

   ```
   aws iam create-policy `
     --policy-name GreengrassSSMAgentComponentPolicy `
     --policy-document file://ssm-agent-component-policy.json
   ```

   出力のポリシーメタデータから、ポリシーの Amazon リソースネーム (ARN) をコピーします。この ARN を使用して、次の手順で、このポリシーをコアデバイスのロールにアタッチします。

3. 次のコマンドを実行して、ポリシーをトークン交換ロールにアタッチします。

   - *GreengrassV2TokenExchangeRole* を、 AWS IoT Greengrass Core ソフトウェアをインストールしたときに指定したトークン交換ロールの名前に置き換えます。AWS IoT

Greengrass Core ソフトウェアのインストール時にトークン交換ロールの名前を指定しなかった場合は、`GreengrassV2TokenExchangeRole` という名前のロールが作成されます。

- ポリシー ARN を、前のステップで書き留めた ARN に置き換えます。

Linux or Unix

```
aws iam attach-role-policy \
  --role-name GreengrassV2TokenExchangeRole \
  --policy-arn
 arn:aws:iam::123456789012:policy/GreengrassSSMAgentComponentPolicy
```

Windows Command Prompt (CMD)

```
aws iam attach-role-policy ^
  --role-name GreengrassV2TokenExchangeRole ^
  --policy-arn
 arn:aws:iam::123456789012:policy/GreengrassSSMAgentComponentPolicy
```

PowerShell

```
aws iam attach-role-policy `
  --role-name GreengrassV2TokenExchangeRole `
  --policy-arn
 arn:aws:iam::123456789012:policy/GreengrassSSMAgentComponentPolicy
```

成功した場合は、コマンドからの出力はありません。これで、このトークン交換ロールを使用するコアデバイスには、Systems Manager サービスとやり取りする権限が付与されます。

ステップ 4: Systems Manager Agent コンポーネントをデプロイする

Systems Manager Agent コンポーネントをデプロイして設定するには、以下のステップを完了します。コンポーネントは 1 つのコアデバイスにも、コアデバイスのグループにもデプロイできます。

Systems Manager Agent コンポーネントをデプロイするには (コンソール)

1. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Components] (コンポーネント) を選択します。

2. [Components] (コンポーネント) ページで、[Public components] (公開コンポーネント) タブを選択し、次に aws.greengrass.SystemsManagerAgent を選択します。
3. aws.greengrass.SystemsManagerAgent ページで、[Deploy] (デプロイ) を選択します。
4. [Add to deployment] (デプロイに追加) で、改訂する既存のデプロイを選択するか、新しいデプロイを作成することを選択して、[Next] (次へ) を選択します。
5. 新しいデプロイの作成を選択した場合、デプロイのターゲットコアデバイスまたはモノグループを選択します。リポジトリの [Specify target] (ターゲットを指定) ページの、[Deployment target] (ターゲットのデプロイ) で、コアデバイスまたはモノグループを選択し、[Next] (次へ) を選択します。
6. [Select components] (コンポーネントを選択) ページで、aws.greengrass.SystemsManagerAgent コンポーネントが選択されていることを確認し、[Next] (次) を選択します。
7. [Configure components] (コンポーネントを設定) ページで、aws.greengrass.SystemsManagerAgent を選択したら、次の操作を行います。

   a. [Configure component] (コンポーネントを設定) を選択します。

   b. [aws.greengrass.SystemsManagerAgent の設定] モーダルの [設定の更新] の下にある [マージする設定] に、次の設定更新を入力します。*SSMServiceRole* を で作成したサービスロールの名前に置き換えます[ステップ 2: Systems Manager の IAM サービスロールを作成する](#)。

```
{
  "SSMRegistrationRole": "SSMServiceRole",
  "SSMOverrideExistingRegistration": false
}
```

> **Note**
>
> コアデバイスで、ハイブリッドアクティベーションに登録されている Systems Manager Agent がすでに実行されている場合は、`SSMOverrideExistingRegistration` を `true` に変更します。このパラメータは、デバイスでハイブリッドアクティベーションが行なわれた Systems Manager Agent がすでに実行されている場合に、Systems Manager Agent コンポーネントがコアデバイスの登録を行うかどうか指定するものです。Systems Manager Agent コンポーネントがコアデバイス用に作成する Systems Manager マネージドノードに追加するタグ (`SSMResourceTags`) を指定することも

できます。詳細については、「Systems Manager Agent コンポーネントの設定」を参照してください。

c. [Confirm] (確認) を選択してモーダルを閉じ、次に [Next] (次) を選択します。

8. [Configure advanced settings] (詳細設定) ページはデフォルト設定のままにし、[Next] (次へ) を選択します。
9. [Review] ページで、[デプロイ] を選択します。

   デプロイに最大 1 分かかる場合があります。

Systems Manager Agent コンポーネントをデプロイするには (AWS CLI)

Systems Manager Agent コンポーネントをデプロイするには、components オブジェクトに aws.greengrass.SystemsManagerAgent を含むデプロイドキュメントを作成し、コンポーネントのコンフィギュレーション更新を指定します。デプロイの作成 の指示に従って、新しいデプロイを作成または既存のデプロイを改訂します。

次の部分的なデプロイドキュメントの例では、SSMServiceRole という名前のサービスロールを使用するように指定しています。*SSMServiceRole* を で作成したサービスロールの名前に置き換えますステップ 2: Systems Manager の IAM サービスロールを作成する。

```
{
  ...,
  "components": {
    ...,
    "aws.greengrass.SystemsManagerAgent": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "merge": "{\"SSMRegistrationRole\":\"SSMServiceRole\",
\"SSMOverrideExistingRegistration\":false}"
      }
    }
  }
}
```

> **Note**
>
> コアデバイスで、ハイブリッドアクティベーションに登録されている Systems Manager Agent がすでに実行されている場合は、SSMOverrideExistingRegistration を true

に変更します。このパラメータは、デバイスでハイブリッドアクティベーションが行なわれた Systems Manager Agent がすでに実行されている場合に、Systems Manager Agent コンポーネントがコアデバイスの登録を行うかどうか指定するものです。
Systems Manager Agent コンポーネントがコアデバイス用に作成する Systems Manager マネージドノードに追加するタグ (`SSMResourceTags`) を指定することもできます。詳細については、「Systems Manager Agent コンポーネントの設定」を参照してください。

デプロイには数分かかる場合があります。AWS IoT Greengrass サービスを使用すると、デプロイのステータスを確認するとともに、AWS IoT Greengrass Core ソフトウェアのログと Systems Manager Agent コンポーネントのログを確認して、Systems Manager Agent が正常に動作していることを確認できます。詳細については、次を参照してください。

- デプロイのステータスを確認する
- AWS IoT Greengrass ログのモニタリング
- 「AWS Systems Manager ユーザーガイド」の「Systems Manager エージェントのログの表示」

デプロイに失敗した場合、または Systems Manager Agent が動作しない場合は、各コアデバイスでデプロイのトラブルシューティングを行います。詳細については、次を参照してください。

- トラブルシューティング AWS IoT Greengrass V2
- 「AWS Systems Manager ユーザーガイド」の「Systems Manager のトラブルシューティング」

ステップ 5: Systems Manager でコアデバイスの登録を検証する

Systems Manager Agent コンポーネントを実行すると、Systems Manager にはコアデバイスがマネージドノードとして登録されます。AWS IoT Greengrass コンソール、Systems Manager コンソール、および Systems Manager API を使用して、コアデバイスがマネージドノードとして登録されていることを確認できます。マネージドノードは、AWS コンソールや API の一部ではインスタンスとも呼ばれます。

コアデバイスの登録を検証するには (AWS IoT Greengrass コンソール)

1. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Core devices] (コアデバイス) を選択します。
2. 検証するコアデバイスを選択します。

3. コアデバイスの詳細ページで、AWS Systems Manager インスタンスプロパティを探します。このプロパティが存在し、Systems Manager コンソールへのリンクが表示されている場合、このコアデバイスはマネージドノードとして登録されています。

   また、AWS Systems Manager ping ステータスプロパティを探して、コアデバイスの Systems Manager Agent のステータスを確認することもできます。ステータスが [Online] (オンライン) のときは、Systems Manager でコアデバイスを管理できます。

コアデバイスの登録を検証するには (Systems Manager コンソール)

1. [Systems Manager console] (Systems Manager コンソール) ナビゲーションメニューで、Fleet Manager を選択します。
2. [Managed nodes] (マネージドノード) で以下を実行します。
   a. [Source type] (ソースタイプ) を AWS::IoT::Thing に設定して、フィルターを追加します。
   b. [Source ID] (ソース ID) を検証するコアデバイスの名前に設定して、フィルターを追加します。
3. [Managed nodes] (マネージドノード) の表でコアデバイスを探します。コアデバイスが表内に見つかった場合、これはマネージドノードとして登録されています。

   また、[Systems Manager Agent ping status] (Systems Manager Agent ping ステータス) プロパティを探して、コアデバイスの Systems Manager Agent のステータスを確認することもできます。ステータスが [Online] (オンライン) のときは、Systems Manager でコアデバイスを管理できます。

コアデバイスの登録を検証するには (AWS CLI)

- DescribeInstanceInformation オペレーションを使用して、指定したフィルターに一致するマネージドノードのリストを取得します。次のコマンドを実行して、コアデバイスがマネージドノードとして登録されているかどうかを確認します。を検証するコアデバイスの名前*MyGreengrassCore*に置き換えます。

  ```
  aws ssm describe-instance-information --filter
   Key=SourceIds,Values=MyGreengrassCore Key=SourceTypes,Values=AWS::IoT::Thing
  ```

  レスポンスには、フィルターに一致するマネージドノードのリストが含まれます。リストにマネージドノードが含まれているなら、コアデバイスはマネージドノードとして登録されていま

す。また、コアデバイスのマネージドノードに関するその他の情報もレスポンスで確認できます。`PingStatus` のプロパティが `Online` であれば、Systems Manager でコアデバイスを管理できます。

コアデバイスがマネージドノードとして登録されていることを Systems Manager で確認した後は、Systems Manager コンソールや API を使用して、そのコアデバイスを管理できます。Greengrass コアデバイスの管理に使用できる Systems Manager の機能の詳細については、「AWS Systems Manager ユーザーガイド」の「Systems Manager の機能」を参照してください。

AWS Systems Manager エージェントのアンインストール

Greengrass コアデバイスを AWS Systems Manager で管理する必要がなくなった場合、Systems Manager からコアデバイスの登録を解除し、デバイスから AWS Systems Manager エージェント (Systems Manager エージェント) をアンインストールできます。

コアデバイスはいつでも再登録できます。これを行うには、Systems Manager エージェントコンポーネントを再度デプロイします。これにより、インストール時にコアデバイスが Systems Manager に登録されます。Systems Manager は、登録解除されたコアデバイスのコマンド履歴を 30 日間保存します。

トピック

- ステップ 1: Systems Manager からコアデバイスを登録解除する
- ステップ 2: Systems Manager エージェントコンポーネントをアンインストールする
- ステップ 3: Systems Manager エージェントソフトウェアをアンインストールする

ステップ 1: Systems Manager からコアデバイスを登録解除する

Systems Manager コンソールまたは API を使用して、コアデバイスの登録を解除できます。詳細については、「AWS Systems Manager ユーザーガイド」の「マネージドノードの登録解除」を参照してください。

ステップ 2: Systems Manager エージェントコンポーネントをアンインストールする

コアデバイスの登録を解除したら、デバイスから [Systems Manager Agent component] (Systems Manager エージェントコンポーネント) をアンインストールします。Greengrass コアデバイスから

コンポーネントを削除するには、コンポーネントをインストールしたデプロイを修正し、デプロイからコンポーネントを削除します。AWS IoT Greengrass Core ソフトウェアは、コアデバイスのいずれのデプロイもそのコンポーネントを指定していない場合、コンポーネントをアンインストールします。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

Systems Manager エージェントコンポーネントをアンインストールするには (コンソール)

1. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Core devices] (コアデバイス) を選択します。
2. Systems Manager エージェントコンポーネントをアンインストールするコアデバイスを選択します。
3. [core device details] (コアデバイスの詳細) ページで、[Deployments] (デプロイ) タブを選択します。
4. Systems Manager エージェントコンポーネントをコアデバイスにデプロイするデプロイを選択します。
5. [deployment details] (デプロイの詳細) ページで、[Revise] (修正) を選択します。
6. [Revise deployment] (デプロイの改訂) モーダルで、[Revise deployment] (デプロイの改訂) を選択します。
7. [Step 1: Specify target] (ステップ 1: ターゲットの指定) で、[Next] (次へ) を選択します。
8. [Step 2: Select components] (ステップ 2: コンポーネントを選択する) で、aws.greengrass.SystemsManagerAgent コンポーネントの選択をクリアしてから、[Next] (次へ) を選択します。
9. [Step 3: Configure components] (ステップ 3: コンポーネントを設定する) で、[Next] (次へ) を選択します。
10. [Step 4: Configure advanced settings] (ステップ 4: 詳細設定を設定する) で、[Next] (次へ) を選択します。
11. [Step 5: Review] (ステップ 5: 確認) で、[Deploy] (デプロイ) を選択します。

Systems Manager エージェントコンポーネントをアンインストールするには (CLI)

Systems Manager エージェントコンポーネントをアンインストールするには、そのコンポーネントをデプロイするデプロイを修正し、デプロイから削除します。詳細については、「展開の改訂」を参照してください。

デプロイには数分かかる場合があります。AWS IoT Greengrass サービスを使用するとデプロイのステータスを確認できます。詳細については、「デプロイのステータスを確認する」を参照してください。

ステップ 3: Systems Manager エージェントソフトウェアをアンインストールする

Systems Manager エージェントコンポーネントを削除した後も、Systems Manager エージェントソフトウェアは、コアデバイスで引き続き実行されます。Systems Manager エージェントソフトウェアを削除するには、コアデバイスでコマンドを実行します。詳細については、「AWS Systems Manager ユーザーガイド」の「Linux インスタンスから Systems Manager エージェントをアンインストールする」を参照してください。

AWS IoT Greengrass のセキュリティ

AWS では、クラウドのセキュリティが最優先事項です。セキュリティを最も重視する組織の要件を満たすために構築された AWS のデータセンターとネットワークアーキテクチャは、お客様に大きく貢献します。

セキュリティは、AWS とお客様とが共有する責務です。責任共有モデル では、これをクラウドのセキュリティおよびクラウド内のセキュリティとして説明しています。

- クラウドのセキュリティ - AWS は、AWS クラウド で AWS のサービスを実行するインフラストラクチャを保護する責任を担います。また、AWS は、ユーザーが安全に使用できるサービスも提供します。AWSコンプライアンスプログラムg11AWSコンプライアンスプログラム/g11g10AWSコンプライアンスプログラム/g10の一環として、サードパーティーの監査が定期的にセキュリティの有効性をテストおよび検証しています。AWS IoT Greengrass に適用するコンプライアンスプログラムの詳細については、「コンプライアンスプログラムによる対象範囲内の AWS のサービス」「」を参照してください。
- クラウド内のセキュリティ - ユーザーの責任は、使用する AWS のサービスに応じて異なります。また、お客様は、お客様のデータの機密性、企業の要件、および適用可能な法律および規制などの他の要因についても責任を担います。

AWS IoT Greengrass を使用する際は、デバイス、ローカルネットワーク接続、およびプライベートキーのセキュリティについてもお客様が責任を負います。

このドキュメントは、AWS IoT Greengrass を使用する際に責任共有モデルを適用する方法を理解するのに役立ちます。以下のトピックでは、セキュリティおよびコンプライアンスの目的を達成するために AWS IoT Greengrass を設定する方法を示します。また、AWS IoT Greengrass リソースのモニタリングや保護に役立つ、その他 AWS サービスの使用方法についても説明します。

トピック

- AWS IoT Greengrass でのインフラストラクチャセキュリティ
- AWS IoT Greengrass での設定と脆弱性の分析
- AWS IoT Greengrass V2 におけるコードの整合性
- AWS IoT Greengrass とインターフェース VPC エンドポイント (AWS PrivateLink)
- AWS IoT Greengrass のセキュリティのベストプラクティス

AWS IoT Greengrass でのデータ保護

AWS責任共有モデルは、AWS IoT Greengrass でのデータ保護に適用されます。このモデルで説明されているように、AWS は、AWS クラウド のすべてを実行するグローバルインフラストラクチャを保護するがあります。お客様は、このインフラストラクチャでホストされているコンテンツに対する管理を維持する責任があります。また、使用する AWS のサービスのセキュリティ設定と管理タスクもユーザーの責任となります。データプライバシーの詳細については、「データプライバシーのよくある質問」を参照してください。欧州でのデータ保護の詳細については、「AWS セキュリティブログ」に投稿された「AWS 責任共有モデルおよび GDPR」のブログ記事を参照してください。

データを保護するため、AWS アカウント の認証情報を保護し、AWS IAM Identity Center または AWS Identity and Access Management (IAM) を使用して個々のユーザーをセットアップすることをお勧めします。この方法により、それぞれのジョブを遂行するために必要な権限のみを各ユーザーに付与できます。また、次の方法でデータを保護することをおすすめします。

- 各アカウントで多要素認証 (MFA) を使用します。
- SSL/TLS を使用して AWS リソースと通信します。TLS 1.2 が必須です。TLS 1.3 が推奨されます。
- AWS CloudTrail で API とユーザーアクティビティロギングをセットアップします。
- AWS のサービス内でデフォルトである、すべてのセキュリティ管理に加え、AWS の暗号化ソリューションを使用します。
- Amazon Macie などの高度なマネージドセキュリティサービスを使用します。これらは、Amazon S3 に保存されている機密データの検出と保護を支援します。
- コマンドラインインターフェイスまたは API により AWS にアクセスするときに FIPS 140-2 検証済み暗号化モジュールが必要な場合は、FIPS エンドポイントを使用します。利用可能な FIPS エンドポイントの詳細については、「連邦情報処理規格 (FIPS) 140-2」を参照してください。

お客様の E メールアドレスなどの極秘または機密情報は、タグ、または名前フィールドなどの自由形式のテキストフィールドに配置しないことを強くお勧めします。これは、コンソール、API、AWS

CLI、または AWS SDK で AWS IoT Greengrass または他の AWS のサービス を使用する場合も同様です。タグ、または名前に使用される自由形式のテキストフィールドに入力されるデータは、請求または診断ログに使用される場合があります。外部サーバーへの URL を提供する場合は、そのサーバーへのリクエストを検証するための認証情報を URL に含めないように強くお勧めします。

AWS IoT Greengrass での機密情報の保護の詳細については、「the section called “機密情報を記録しない”」を参照してください。

データ保護の詳細については、AWSセキュリティブログ のブログ投稿「AWSの責任共有モデルとGDPR」を参照してください。

トピック

- データ暗号化
- ハードウェアセキュリティ統合

データ暗号化

AWS IoT Greengrass は暗号化を使用して、(インターネットまたはローカルネットワークで) 転送中および保管中の (AWS クラウド に保存されている) データを保護します。

AWS IoT Greengrass 環境にあるデバイスは、多くの場合、処理のために AWS のサービスに送信されるデータを収集します。他の AWS のサービスのデータ暗号化の詳細については、そのサービスのセキュリティドキュメントを参照してください。

トピック

- 転送中の暗号化
- 保管中の暗号化
- Greengrass Core Device のキー管理

転送中の暗号化

AWS IoT Greengrass には、データが転送中の 2 つの通信モードがあります。

- the section called “インターネット経由で転送されるデータ”. インターネットを介した Greengrass コアと AWS IoT Greengrass 間の通信は暗号化されます。
- the section called “コアデバイス上のデータ”: Greengrass コアデバイス上のコンポーネント間の通信は暗号化されません。

インターネット経由で転送されるデータ

AWS IoT Greengrass は、Transport Layer Security (TLS) を使用して、インターネット経由のすべての通信を暗号化します。AWS クラウド に送信されるすべてのデータは、MQTT または HTTPS プロトコルを使用して TLS 接続で送信されるため、デフォルトで安全に保護されています。AWS IoT Greengrass は、AWS IoT トランスポートセキュリティモデルを使用します。詳細については、[AWS IoT Core Developer Guide] (デベロッパーガイド) の[Transport security] (トランスポートセキュリティ) を参照してください。

コアデバイス上のデータ

AWS IoT Greengrass は、Greengrass コアデバイス上でローカルに交換されたデータを暗号化しません。これは、データがデバイスから離れることがないためです。これには、ユーザー定義コンポーネント間の通信、AWS IoT デバイスの SDK、およびストリームマネージャーなどのパブリックコンポーネントが含まれます。

保管中の暗号化

AWS IoT Greengrass は、データを保存します。

- the section called “AWS クラウド クラウドに保管中のデータ”. このデータは暗号化されます。
- the section called “Greengrass コアに保管されているデータ”. このデータは暗号化されません (シークレットのローカルコピーを除きます)。

AWS クラウド クラウドに保管中のデータ

AWS IoT Greengrass は、AWS クラウド 内に保管中のお客様データを暗号化します。このデータは、AWS IoT Greengrass によって管理される AWS KMS キーを使用して保護されます。

Greengrass コアに保管されているデータ

AWS IoT Greengrass は、Unix ファイルアクセス許可とフルディスク暗号化 (有効になっている場合) に依存して、コアに保管されているデータを保護します。ファイルシステムとデバイスを保護するのはお客様の責任となります。

ただし、AWS IoT Greengrass は AWS Secrets Manager から取得したシークレットのローカルコピーを暗号化します。詳細については、Secrets Manager コンポーネントを参照してください。

Greengrass Core Device のキー管理

Greengrass Core Device の暗号化 (パブリックおよびプライベート) キーを安全に保管するのは、お客様の責任です。AWS IoT Greengrass は次のシナリオでパブリックキーおよびプライベートキーを使用します。

- IoT クライアントキーは IoT 証明書とともに使用され、Greengrass Core が AWS IoT Core に接続すると Transport Layer Security (TLS) ハンドシェイクを認証します。詳細については、「the section called “デバイス認証と認可”」を参照してください。

> **Note**
>
> キーおよび証明書は、コアプライベートキーおよびコアデバイス証明書とも呼ばれます。

Greengrass Core デバイスは、ファイルシステムのアクセス許可、またはハードウェアセキュリティモジュールを使用して、プライベートキーストレージをサポートします。ファイルシステムベースのプライベートキーを使用する場合は、お客様がコアデバイス上の安全な保管の責任を負います。

ハードウェアセキュリティ統合

> **Note**
>
> この機能は、Greengrass nucleus コンポーネントの v2.5.3 以降に利用できます。AWS IoT Greengrass は、現在 Windows コアデバイスにこの機能をサポートしていません。

AWS IoT Greengrass Core ソフトウェアは、PKCS#11 インターフェイスを介してハードウェアセキュリティモジュール (HSM) を使用するように設定できます。この機能を使用すると、デバイスのプライベートキーと証明書を安全に保存して、ソフトウェアで公開または複製されないようにできます。プライベートキーと証明書は、HSM やトラステッドプラットフォームモジュール (TPM) などのハードウェアモジュールに保存できます。

AWS IoT Greengrass Core ソフトウェアは、プライベートキーと X.509 証明書を使用して、AWS IoT や AWS IoT Greengrass のサービスへの接続を認証します。シークレットマネージャーコンポーネントは、このプライベートキーを使用することで、Greengrass コアデバイスにデプロイするシークレットの暗号化と復号化を安全に行うことができます。HSM を使用するようにコアデバイスを設定する場合、これらのコンポーネントは HSM に保存したプライベートキーと証明書を使用します。

Moquette MQTT ブローカーコンポーネントも、ローカルの MQTT サーバー証明書のプライベートキーを格納します。このコンポーネントは、プライベートキーをデバイスのファイルシステム上のコンポーネントのワークフォルダに保存します。現在 AWS IoT Greengrass では、このプライベートキーまたは証明書を HSM に格納することをサポートしていません。

> **Tip**
>
> この機能をサポートするデバイスを検索するには、「AWS Partner Device Catalog」を参照してください。

トピック

要件

Greengrass コアデバイスで HSM を使用するには、以下の要件を満たしている必要があります。

- Greengrass nucleus v2.5.3 以降がコアデバイスにインストールされている。コアデバイスに AWS IoT Greengrass Core ソフトウェアをインストールする際に、互換性のあるバージョンを選択できます。
- PKCS#11 プロバイダコンポーネントがコアデバイスにインストールされている。コアデバイスに AWS IoT Greengrass Core ソフトウェアをインストールする際に、このコンポーネントをダウンロードしてインストールできます。
- PKCS#1 v1.5 署名スキームと RSA-2048 キーサイズ (またはそれ以上の規模) または ECC キーを備えた RSA キーをサポートするハードウェアセキュリティモジュール。

> **Note**
>
> ECC キーを備えたハードウェアセキュリティモジュールを使用するには、v2.5.6 以降の Greengrass nucleus を使用する必要があります。

ハードウェアセキュリティモジュールとシークレットマネージャーを使用するには、RSA キーを備えたハードウェアセキュリティモジュールを使用する必要があります。

- PKCS#11 関数を呼び出すため、AWS IoT Greengrass Core ソフトウェアがランタイム時 (libdl を使用) にロードできる PKCS#11 プロバイダライブラリ。PKCS#11 プロバイダライブラリは、次の PKCS#11 API オペレーションを実装する必要があります。
  - `C_Initialize`
  - `C_Finalize`
  - `C_GetSlotList`
  - `C_GetSlotInfo`
  - `C_GetTokenInfo`
  - `C_OpenSession`
  - `C_GetSessionInfo`
  - `C_CloseSession`
  - `C_Login`
  - `C_Logout`
  - `C_GetAttributeValue`
  - `C_FindObjectsInit`
  - `C_FindObjects`
  - `C_FindObjectsFinal`
  - `C_DecryptInit`
  - `C_Decrypt`
  - `C_DecryptUpdate`
  - `C_DecryptFinal`
  - `C_SignInit`
  - `C_Sign`
  - `C_SignUpdate`
  - `C_SignFinal`
  - `C_GetMechanismList`
  - `C_GetMechanismInfo`
  - `C_GetInfo`

  - `C_GetFunctionList`
- ハードウェアモジュールは、「PKCS#11 仕様」で定義されているスロットラベルで解決できる必要があります。
- プライベートキーと証明書は HSM の同じスロットに保存する必要があり、HSM がオブジェクト ID をサポートしている場合、同じオブジェクトラベルとオブジェクト ID を使用する必要があります。
- 証明書とプライベートキーがオブジェクトラベルで解決できる必要があります。
- プライベートキーには、次の許可が必要です。
  - `sign`
  - `decrypt`
- (オプション) シークレットマネージャーコンポーネントを使用する場合、バージョン 2.1.0 以降を使用する必要があります。また、プライベートキーには次の許可が必要です。
  - `unwrap`
  - `wrap`

ハードウェアセキュリティのベストプラクティス

Greengrass コアデバイスでハードウェアセキュリティを設定する場合は、以下のベストプラクティスを考慮してください。

- 内部ハードウェア乱数ジェネレーターを使用して、HSM に直接プライベートキーを生成します。この方法では、プライベートキーが HSM 内に残るため、他の場所で生成したプライベートキーをインポートするよりも安全です。
- プライベートキーを変更不可およびエクスポートを禁止するように設定します。
- HSM ハードウェアベンダーが推奨するプロビジョニングツールを使用して、ハードウェアで保護されたプライベートキーにより証明書署名リクエスト (CSR) を生成します。次に、AWS IoT コンソールを使用してクライアント証明書を生成します。

> **Note**
>
> HSM でプライベートキーを生成する場合は、キーをローテーションするセキュリティ上のベストプラクティスは適用されません。

ハードウェアセキュリティを備えた AWS IoT Greengrass Core ソフトウェアのインストール

AWS IoT Greengrass Core ソフトウェアをインストールする場合、HSM で生成したプライベートキーを使用するように設定できます。このアプローチは、HSM でプライベートキーを生成するためのセキュリティ上のベストプラクティスに従うため、プライベートキーは HSM 内に残ります。

ハードウェアセキュリティを備えた AWS IoT Greengrass Core ソフトウェアをインストールするには、次の操作を行います。

1. HSM でプライベートキーを生成します。
2. プライベートキーから証明書署名リクエスト (CSR) を作成します。
3. CSR から証明書を作成します。AWS IoT または、別のルート認証機関 (CA) によって署名された証明書を作成できます。別のルート CA の使用方法の詳細については、「AWS IoT Core デベロッパーガイド」の「独自のクライアント証明書を作成する」を参照してください。
4. AWS IoT の証明書をダウンロードして、HSM にインポートします。
5. HSM で PKCS#11 プロバイダコンポーネントとプライベートキーおよび証明書を使用するように指定する設定ファイルから AWS IoT Greengrass Core ソフトウェアをインストールします。

ハードウェアセキュリティを備えたAWS IoT Greengrass Core ソフトウェアをインストールするには、次のインストールオプションのいずれかを選択します。

- 手動インストール

  必要な AWS リソースを作成し、ハードウェアセキュリティを手動で作成するには、このオプションを選択します。詳細については、「手動リソースプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。
- カスタムプロビジョニングを使用したインストール

  必要な AWS リソースを作成し、ハードウェアセキュリティを自動的に作成するカスタムの Java アプリケーションを開発するには、このオプションを選択します。詳細については、「カスタムリソースプロビジョニングを使用して AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。

現在 AWS IoT Greengrass では、自動リソースプロビジョニングによるインストールまたはAWS IoT フリートプロビジョニングを行う際の、ハードウェアセキュリティを備えた AWS IoT Greengrass Core ソフトウェアのインストールはサポートされていません。

既存のコアデバイスでのハードウェアセキュリティの設定

コアデバイスのプライベートキーと証明書を HSM にインポートして、ハードウェアセキュリティを設定できます。

> **考慮事項**
>
> - コアデバイスのファイルシステムへのルートアクセス権が必要です。
> - この手順では、AWS IoT Greengrass Core ソフトウェアをシャットダウンし、ハードウェアセキュリティの設定中はコアデバイスがオフラインになり、利用できなくなります。

既存のコアデバイスでハードウェアセキュリティを設定するには、次の手順を実行します。

1. HSM を初期化します。
2. PKCS#11 プロバイダコンポーネントをコアデバイスにデプロイします。
3. AWS IoT Greengrass Core ソフトウェアを停止します。
4. コアデバイスのプライベートキーと証明書を HSM にインポートします。
5. AWS IoT Greengrass Core ソフトウェアの設定ファイルを、HSM でプライベートキーと証明書を使用するように更新します。
6. AWS IoT Greengrass Core ソフトウェアを起動します。

ステップ 1: ハードウェアセキュリティモジュールを初期化する

次の手順を実行して、コアデバイスで HSM を初期化します。

ハードウェアセキュリティモジュールを初期化するには

- HSM で PKCS#11 トークンを初期化し、トークンのスロット ID とユーザー PIN を保存します。トークンの初期化方法については、HSM のドキュメントを参照してください。スロット ID とユーザー PIN は、後で PKCS#11 プロバイダーコンポーネントをデプロイして設定するときに使用します。

ステップ 2: PKCS#11 プロバイダコンポーネントをデプロイする

PKCS#11 プロバイダコンポーネントをデプロイして設定するには、以下の操作を行います。コンポーネントは 1 つ以上のコアデバイスに展開できます。

PKCS#11 プロバイダコンポーネントをデプロイするには (コンソール)

1. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Components] (コンポーネント) を選択します。
2. [Components] (コンポーネント) ページで、[Public components] (公開コンポーネント) タブを選択し、次に aws.greengrass.crypto.Pkcs11Provider を選択します。
3. aws.greengrass.crypto.Pkcs11Provider ページで、[Deploy] (デプロイ) を選択します。
4. [Add to deployment] (デプロイに追加) で、改訂する既存のデプロイを選択するか、新しいデプロイを作成することを選択して、[Next] (次へ) を選択します。
5. 新しいデプロイの作成を選択した場合、デプロイのターゲットコアデバイスまたはモノグループを選択します。リポジトリの [Specify target] (ターゲットを指定) ページの、[Deployment target] (ターゲットのデプロイ) で、コアデバイスまたはモノグループを選択し、[Next] (次へ) を選択します。
6. [Select components] (コンポーネントを選択) ページの[Public components] (パブリックコンポーネント) から、[aws.greengrass.crypto.Pkcs11Provider] を選択してから、[Next] (次へ) をクリックします。
7. [Configure components] (コンポーネントを設定) ページで、aws.greengrass.crypto.Pkcs11Provider を選択したら、次の操作を行います。

   a. [Configure component] (コンポーネントを設定) を選択します。
   b. [aws.greengrass.crypto.Pkcs11Provider の設定] モーダルの [設定の更新] の下にある [マージする設定] に、次の設定更新を入力します。ターゲットコアデバイスの値を使用して、次の設定パラメータを更新します。PKCS#11 トークンを初期化したスロット ID とユーザー PIN を指定します。プライベートキーと証明書は、後で HSM のこのスロットにインポートします。

      `name`

      PKCS#11 設定の名前。

      `library`

      AWS IoT Greengrass Core ソフトウェアが libdl でロードできる PKCS#11 実装のライブラリへの絶対ファイルパス

      `slot`

      プライベートキーとデバイス証明書を含むスロットの ID。この値は、スロットインデックスやスロットラベルとは異なります。

`userPin`

スロットへのアクセスに使用するユーザー PIN。

```
{
  "name": "softhsm_pkcs11",
  "library": "/usr/lib/softhsm/libsofthsm2.so",
  "slot": 1,
  "userPin": "1234"
}
```

c. [Confirm] (確認) を選択してモーダルを閉じ、次に [Next] (次) を選択します。

8. [Configure advanced settings] (詳細設定) ページはデフォルト設定のままにし、[Next] (次へ) を選択します。
9. [Review] ページで、[デプロイ] を選択します。

   デプロイに最大 1 分かかる場合があります。

PKCS#11 プロバイダコンポーネントをデプロイするには (AWS CLI)

PKCS#11 プロバイダコンポーネントをデプロイするには、`components` オブジェクトの `aws.greengrass.crypto.Pkcs11Provider` を含むデプロイドキュメントを作成し、コンポーネントのコンフィギュレーション更新を指定します。デプロイの作成 の指示に従って、新しいデプロイを作成または既存のデプロイを改訂します。

次の部分デプロイドキュメントの例では、PKCS#11 プロバイダコンポーネントのデプロイと設定を行うよう指定しています。ターゲットコアデバイスの値を使用して、次の設定パラメータを更新します。後でプライベートキーと証明書を HSM にインポートする際に使用するスロット ID とユーザー PIN を保存します。

`name`

PKCS#11 設定の名前。

`library`

AWS IoT Greengrass Core ソフトウェアが libdl でロードできる PKCS#11 実装のライブラリへの絶対ファイルパス

`slot`

プライベートキーとデバイス証明書を含むスロットの ID。この値は、スロットインデックスやスロットラベルとは異なります。

`userPin`

スロットへのアクセスに使用するユーザー PIN。

```
{
  "name": "softhsm_pkcs11",
  "library": "/usr/lib/softhsm/libsofthsm2.so",
  "slot": 1,
  "userPin": "1234"
}
```

```
{
  ...,
  "components": {
    ...,
    "aws.greengrass.crypto.Pkcs11Provider": {
      "componentVersion": "2.0.0",
      "configurationUpdate": {
        "merge": "{\"name\":\"softhsm_pkcs11\",\"library\":\"/usr/lib/softhsm/
libsofthsm2.so\",\"slot\":1,\"userPin\":\"1234\"}"
      }
    }
  }
}
```

デプロイには数分かかる場合があります。AWS IoT Greengrass サービスを使用するとデプロイのステータスを確認できます。PKCS#11 プロバイダコンポーネントが正常にデプロイされたことを検証するには、AWS IoT Greengrass Core ソフトウェアログを確認します。詳細については、次を参照してください。

- デプロイのステータスを確認する
- AWS IoT Greengrass ログのモニタリング

デプロイに失敗した場合は、各コアデバイスでデプロイのトラブルシューティングを行います。詳細については、「トラブルシューティング AWS IoT Greengrass V2」を参照してください。

ステップ 3: コアデバイスの設定を更新する

AWS IoT Greengrass Core ソフトウェアでは、デバイスの動作を指定するコンフィギュレーションファイルを使用します。この設定ファイルには、デバイスが AWS クラウド に接続するために使用するプライベートキーおよび証明書の所在が含まれています。次の手順を実行して、コアデバイスのプライベートキーと証明書を HSM にインポートし、HSM を使用するように設定ファイルを更新します。

ハードウェアセキュリティを使用するようにコアデバイスの設定を更新するには

1. AWS IoT Greengrass Core ソフトウェアを停止します。例えば systemd で AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定 した場合は、次のコマンドを実行してソフトウェアを停止できます。

   ```
   sudo systemctl stop greengrass.service
   ```

2. コアデバイスのプライベートキーと証明書ファイルを検索します。

   - 自動プロビジョニングまたはフリートプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールした場合は、プライベートキーは */greengrass/v2*/privKey.key にあり、証明書は */greengrass/v2*/thingCert.crt にあります。
   - 手動プロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールした場合、プライベートキーは、デフォルトで */greengrass/v2*/private.pem.key にあり、証明書はデフォルトで */greengrass/v2*/device.pem.crt にあります。

   /greengrass/v2/config/effectiveConfig.yaml の system.privateKeyPath と system.certificateFilePath のプロパティで、これらのファイルの場所を確認します。

3. プライベートキーと証明書を HSM にインポートします。プライベートキーと証明書をインポートする方法については、HSM のドキュメントを参照してください。PKCS#11 トークンを初期化したスロット ID とユーザー PIN を使用して、プライベートキーと証明書をインポートします。プライベートキーと証明書には、同じオブジェクトラベルとオブジェクト ID を使用する必要があります。各ファイルをインポートしたときに指定したオブジェクトラベルを保存します。HSM でプライベートキーと証明書を使用するように、後で AWS IoT Greengrass Core ソフトウェアの設定を更新する際にこのラベルを使用します。
4. AWS IoT Greengrass Core を HSM でプライベートキーと証明書を使用するように更新します。設定を更新するには、AWS IoT Greengrass Core 設定ファイルを修正し、更新した設定ファイルを用いて AWS IoT Greengrass Core ソフトウェアを実行して、新しい設定を適用します。

以下の操作を実行します。

a. AWS IoT Greengrass Core 設定ファイルのバックアップを作成します。ハードウェアセキュリティの設定時に問題が発生した場合は、このバックアップを使用してコアデバイスを復元できます。

```
sudo cp /greengrass/v2/config/effectiveConfig.yaml ~/ggc-config-backup.yaml
```

b. AWS IoT Greengrass Core 設定ファイルをテキストエディタで開きます。例えば、次のコマンドを実行して GNU nano を使用してファイルを編集できます。を Greengrass ルートフォルダへのパス*/greengrass/v2*に置き換えます。

```
sudo nano /greengrass/v2/config/effectiveConfig.yaml
```

c. `system.privateKeyPath` の値を HSM のプライベートキーの PKCS#11 URI に置き換えます。*iotdeviceKey* を以前にプライベートキーと証明書をインポートしたオブジェクトラベルに置き換えます。

```
pkcs11:object=iotdevicekey;type=private
```

d. `system.certificateFilePath` の値を HSM の証明書の PKCS#11 URI に置き換えます。*iotdeviceKey* を以前にプライベートキーと証明書をインポートしたオブジェクトラベルに置き換えます。

```
pkcs11:object=iotdevicekey;type=cert
```

これらの手順を完了すると、AWS IoT Greengrass Core 設定ファイルの `system` のプロパティは次の例のようになります。

```
system:
  certificateFilePath: "pkcs11:object=iotdevicekey;type=cert"
  privateKeyPath: "pkcs11:object=iotdevicekey;type=private"
  rootCaPath: "/greengrass/v2/rootCA.pem"
  rootpath: "/greengrass/v2"
  thingName: "MyGreengrassCore"
```

5. 更新済みの effectiveConfig.yaml ファイルの設定を適用します。Greengrass.jar を --init-config パラメータで実行して、effectiveConfig.yaml の設定を適用します。を Greengrass ルートフォルダへのパス*/greengrass/v2*に置き換えます。

```
sudo java -Droot="/greengrass/v2" \
  -jar /greengrass/v2/alts/current/distro/lib/Greengrass.jar \
  --start false \
  --init-config /greengrass/v2/config/effectiveConfig.yaml
```

6. AWS IoT Greengrass Core ソフトウェアを起動します。systemd で AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定した場合は、次のコマンドを実行してソフトウェアを起動できます。

```
sudo systemctl start greengrass.service
```

詳細については、「AWS IoT Greengrass Core ソフトウェアを実行する」を参照してください。

7. AWS IoT Greengrass Core ソフトウェアログで、ソフトウェアが起動し、AWS クラウド と接続されることを確認します。AWS IoT Greengrass Core ソフトウェアは、プライベートキーと証明書を使用して、AWS IoT と AWS IoT Greengrass のサービスに接続します。

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

次の情報レベルのログメッセージは、AWS IoT Greengrass Core ソフトウェアが AWS IoT と AWS IoT Greengrass のサービスに正常に接続されていることを示します。

```
2021-12-06T22:47:53.702Z [INFO] (Thread-3)
 com.aws.greengrass.mqttclient.AwsIotMqttClient: Successfully connected to AWS IoT
 Core. {clientId=MyGreengrassCore5, sessionPresent=false}
```

8. (オプション) AWS IoT Greengrass Core ソフトウェアが HSM のプライベートキーと証明書で動作することを確認したら、デバイスのファイルシステムからプライベートキーと証明書ファイルを削除します。次のコマンドを実行し、ファイルパスをプライベートキーおよび証明書ファイルへのパスに置き換えます。

```
sudo rm /greengrass/v2/privKey.key
sudo rm /greengrass/v2/thingCert.crt
```

PKCS#11 をサポートしないハードウェア

PKCS#11 ライブラリは通常、ハードウェアベンダーによって提供されるか、オープンソースです。例えば、標準準拠のハードウェア (TPM1.2 など) では、既存のオープンソースソフトウェアを使用できます。ただし、ハードウェアに対応する PKCS#11 ライブラリ実装がない場合、またはカスタム PKCS#11 プロバイダを作成する場合、統合については Amazon Web Service エンタープライズサポート担当者までお問い合わせください。

以下も参照してください。

- PKCS #11 Cryptographic Token Interface Usage Guide Version 2.4.0 (PKCS #11 暗号トークンインターフェース使用ガイド バージョン 2.4.0)
- RFC 7512
- PKCS #1: RSA Encryption Version 1.5

AWS IoT Greengrass のデバイス認証と認可

AWS IoT Greengrass 環境にあるデバイスは、認証に X.509 証明書を使用し、認可に AWS IoT ポリシーを使用します。証明書とポリシーにより、デバイスは、AWS IoT Core と AWS IoT Greengrass に安全に接続できます。

X.509 証明書は、X.509 パブリックキーインフラストラクチャ規格を使用して、パブリックキーと証明書内の ID を関連付けるための、デジタル証明書です。X.509 証明書は、証明機関 (CA) と呼ばれる信頼された団体によって発行されます。CA は、CA 証明書と呼ばれる 1 つ以上の特別な証明書を管理しており、この証明書は X.509 証明書を発行するために使用されます。証明機関にのみ CA 証明書に対するアクセス権限があります。

AWS IoT ポリシーは、AWS IoT デバイスに対して許可される一連のオペレーションを定義します。具体的には、MQTT メッセージの公開やデバイスシャドウの取得など、AWS IoT Core および AWS IoT Greengrass データプレーンオペレーションに対するアクセスを許可するか拒否します。

すべてのデバイスには、AWS IoT Core レジストリのエントリと、AWS IoT ポリシーがアタッチされたアクティブ化された X.509 証明書が必要です。デバイスは、次の 2 つのカテゴリに分類されます。

- Greengrass コアデバイス

Greengrass コアデバイスは、証明書と AWS IoT ポリシーを使用して AWS IoT Core および AWS IoT Greengrass に安全に接続します。また、証明書とポリシーにより、AWS IoT Greengrass はコアデバイスにコンポーネントや設定をデプロイすることができます。

- クライアントデバイス

  MQTT クライアントデバイスは、証明書とポリシーを使用して、AWS IoT Core および AWS IoT Greengrass サービスに接続します。これにより、クライアントデバイスは AWS IoT Greengrass クラウドディスカバリサービスを使用して、Greengrass コアデバイスを検索して接続することができます。クライアントデバイスは、同じ証明書を使用して AWS IoT Core クラウドサービスとコアデバイスに接続します。また、クライアントデバイスは、コアデバイスとの相互認証に検出情報を使用します。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

X.509 証明書

コアデバイスとクライアントデバイス間の通信、およびデバイスと AWS IoT Core または AWS IoT Greengrass 間の通信は、認証されている必要があります。この相互認証は、登録された X.509 デバイス証明書と暗号化キーに基づいています。

AWS IoT Greengrass 環境では、デバイスは、次の Transport Layer Security (TLS) 接続に対して、パブリックキーとプライベートキーを持つ証明書を使用します。

- インターネット上で AWS IoT Core および AWS IoT Greengrass に接続する Greengrass コアデバイスの AWS IoT クライアントコンポーネント。
- インターネット経由で AWS IoT Greengrass に接続し、コアデバイスを検出するクライアントデバイス。
- ローカルネットワークを介してグループ内の Greengrass デバイスに接続する Greengrass コアにある MQTT ブローカーコンポーネント。

AWS IoT Greengrass コアデバイスは Greengrass ルートフォルダに証明書を格納します。

認証機関 (CA) 証明書

Greengrass コアデバイスとクライアントデバイスが、AWS IoT Core および AWS IoT Greengrass サービスとの認証に使用されるルート CA 証明書をダウンロードします。Amazon ルート CA 1 など、Amazon Trust Services (ATS) のルート CA 証明書を使用することをお勧めします。詳細については、「AWS IoT Core デベロッパーガイド」の「サーバー認証用の CA 証明書」を参照してください。

クライアントデバイスは Greengrass コアデバイスの CA 証明書もダウンロードします。この証明書は、相互認証中にコアデバイスにある MQTT サーバー証明書を検証するために使用されます。

ローカル MQTT ブローカー上での証明書ローテーション

クライアントデバイスのサポートを有効にすると、Greengrass コアデバイスは、クライアントデバイスが相互認証のために使用するローカル MQTT サーバー証明書を生成します。証明書は、コアデバイスが AWS IoT Greengrass クラウドに保存するコアデバイス CA 証明書によって署名されます。クライアントデバイスは、コアデバイスを検出したときにコアデバイスの CA 証明書を取得します。コアデバイスの CA 証明書は、コアデバイスに接続するときに、コアデバイスの MQTT サーバー証明書を検証するために使用します。コアデバイスの CA 証明書は 5 年後に期限切れになります。

MQTT サーバー証明書はデフォルトで 7 日ごとに期限切れとなりますが、この期間は 2～10 日の間に設定できます。この制限期間は、セキュリティのベストプラクティスに基づいています。このローテーションは、攻撃者が MQTT サーバー証明書と秘密キーを盗んで Greengrass コアデバイスを偽装する脅威を軽減するために役立ちます。

Greengrass コアデバイスは、有効期限が切れる 24 時間前に、MQTT サーバー証明書をローテーションします。Greengrass コアデバイスは新しい証明書を生成し、ローカル MQTT ブローカーを再起動します。これが実行されると、Greengrass コアデバイスに接続されているすべてのクライアントデバイスが切断されます。クライアントデバイスは、短時間待機した後に、Greengrass コアデバイスに再度接続できます。

データプレーンオペレーションの AWS IoT ポリシー

AWS IoT ポリシーを使用して、AWS IoT Core および AWS IoT Greengrass データプレーンへのアクセスを許可します。AWS IoT Core データプレーンは、デバイス、ユーザー、およびアプリケーションへの操作を提供します。これらの操作には、AWS IoT Core に接続し、トピックを購読する機能が含まれます。AWS IoT Greengrass データプレーンは Greengrass デバイスへの操作を提供します。詳細については、「AWS IoT Greengrass V2 ポリシーアクション」を参照してください。これらの操作には、コンポーネントの依存関係を解決し、パブリックコンポーネントのアーティファクトをダウンロードする機能が含まれます。

AWS IoT ポリシーは、IAM ポリシーに似た JSON ドキュメントです。これには、次のプロパティを指定する 1 つ以上のポリシーステートメントが含まれます。

- `Effect`。アクセスモードを指定するプロパティで、`Allow` か `Deny` のどちらかになります。
- `Action`。ポリシーによって許可または拒否されるアクションのリストです。
- `Resource`。アクションが許可または拒否されるリソースのリストです。

AWS IoT ポリシーでは * をワイルドカード文字としてサポートし、MQTT ワイルドカード文字 (+ および #) をリテラル文字列として処理します。「*」ワイルドカードの詳細については、「AWS Identity and Access Management ユーザーガイド」の「リソース ARN でのワイルドカードの使用」を参照してください。

詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT ポリシー」と「AWS IoT ポリシーアクション」を参照してください。

> **Important**
>
> モノのポリシー変数 (`iot:Connection.Thing.*`) は コアデバイスまたは Greengrass データプレーン操作用の AWS IoT ポリシーではサポートされていません。代わりに、ワイルドカードを使用して名前が似ている複数のデバイスと一致させることができます。たとえば、`MyGreengrassDevice*` と指定すると `MyGreengrassDevice1`、`MyGreengrassDevice2` などと一致します。

> **Note**
>
> AWS IoT Core では AWS IoT ポリシーをモノのグループにアタッチして、デバイスのグループの権限を定義することができます。モノのグループポリシーでは、AWS IoT Greengrass データプレーンオペレーションへのアクセスは許可されていません。モノが AWS IoT Greengrass のデータプレーンオペレーションにアクセスするには、モノの証明書にアタッチする AWS IoT ポリシーにアクセス許可を追加する必要があります。

AWS IoT Greengrass V2 ポリシーアクション

AWS IoT Greengrass V2 は、Greengrass コアデバイスとクライアントデバイスが AWS IoT ポリシーで使用できる次のポリシーアクションを定義します。ポリシーアクションのリソースを指定するには、リソースの Amazon リソースネーム (ARN) を使用することができます。

コアデバイスのアクション

`greengrass:GetComponentVersionArtifact`

パブリックコンポーネントアーティファクトまたは Lambda コンポーネントアーティファクトのダウンロードに使用する、署名付き URL を取得するアクセス許可を付与します。

このアクセス許可は、コアデバイスがパブリックコンポーネントまたはアーティファクトを持つ Lambda を指定するデプロイを受信したときに評価されます。コアデバイスにアーティファクトがすでに存在する場合、アーティファクトが再度ダウンロードされることはありません。

リソースタイプ: `componentVersion`

リソース ARN 形式: `arn:aws:greengrass:`*`region`*`:`*`account-id`*`:components:`*`component-name`*`:versions:`*`component-version`*

`greengrass:ResolveComponentCandidates`

デプロイのコンポーネント、バージョン、およびプラットフォームの要件を満たすコンポーネントのリストを特定するためのアクセス許可を付与します。要件が競合する場合、または要件を満たすコンポーネントが存在しない場合は、この操作はエラーを返し、デバイスでのデプロイが失敗します。

このアクセス許可は、コアデバイスがコンポーネントを指定するデプロイを受信したときに評価されます。

リソースタイプ: なし

リソース ARN 形式: *

`greengrass:GetDeploymentConfiguration`

大規模なデプロイドキュメントをダウンロードするための署名付き URL を取得するアクセス許可を付与します。

このアクセス許可は、コアデバイスが 7 KB (デプロイがモノをターゲットとする場合) または 31 KB (デプロイがモノグループをターゲットとする場合) を超えるデプロイを指定しているデプロイを受信したときに評価されます。デプロイドキュメントには、コンポーネント設定、デプロイポリシー、およびデプロイメタデータが含まれます。詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

この機能は、Greengrass nucleus コンポーネントの v2.3.0 以降に利用できます。

リソースタイプ: なし

リソース ARN 形式: *

`greengrass:ListThingGroupsForCoreDevice`

コアデバイスのモノグループ階層を取得するアクセス許可を付与します。

このアクセス許可は、コアデバイスが AWS IoT Greengrass からデプロイを受信するときに確認されます。コアデバイスはこのアクションを使用して、前回のデプロイ以降にモノグループから削除されていないかどうかを識別します。コアデバイスがモノグループから削除されており、そのモノグループがコアデバイスへのデプロイターゲットである場合、コアデバイスはそのデプロイでインストールされたコンポーネントを削除します。

この機能は、Greengrass nucleus コンポーネントの v2.5.0 以降で使用されます。

リソースタイプ: `thing` (コアデバイス)

リソース ARN 形式: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`core-device-thing-name`*

`greengrass:VerifyClientDeviceIdentity`

コアデバイスに接続するクライアントデバイスの ID を確認するためのアクセス許可を付与します。

このアクセス許可は、コアデバイスがクライアントデバイス認証コンポーネントを実行し、クライアントデバイスから MQTT 接続を受信したときに評価されます。クライアントデバイスは、この AWS IoT デバイス証明書を提示します。その後、コアデバイスはクライアントデバイスの ID を検証するため、デバイス証明書を AWS IoT Greengrass クラウドサービスに送信します。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

リソースタイプ: なし

リソース ARN 形式: *

`greengrass:VerifyClientDeviceIoTCertificateAssociation`

クライアントデバイスが AWS IoT 証明書と関連付けられているかどうかを検証するためのアクセス許可を付与します。

このアクセス許可は、コアデバイスがクライアントデバイス認証コンポーネントを実行し、クライアントデバイスに MQTT 経由での接続を承認したときに評価されます。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

> **Note**
>
> コアデバイスがこの操作を使用するには、Greengrass サービスロールを AWS アカウント に関連づけ、`iot:DescribeCertificate` の権限を許可する必要があります。

リソースタイプ: thing (クライアントデバイス)

リソース ARN 形式: arn:aws:iot:*region*:*account-id*:thing/*client-device-thing-name*

greengrass:PutCertificateAuthorities

コアデバイスを検証するためにクライアントデバイスがダウンロードできる認証機関 (CA) 証明書をアップロードするためのアクセス許可を付与します。

このアクセス許可は、コアデバイスがクライアントデバイス認証コンポーネントをインストールして実行したときに評価されます。このコンポーネントは、ローカル認証機関を作成し、この操作を使用して CA 証明書をアップロードします。クライアントデバイスは、Discover 操作を使用して接続可能なコアデバイスを検出するときに、これらの CA 証明書をダウンロードします。クライアントデバイスがコアデバイス上の MQTT ブローカーに接続するときに、これらの CA 証明書を使用してコアデバイスの ID を検証します。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

リソースタイプ: なし

ARN 形式: *

greengrass:GetConnectivityInfo

コアデバイスの接続情報を取得するアクセス許可を付与します。この情報は、クライアントデバイスがコアデバイスに接続する方法を説明するものです。

このアクセス許可は、コアデバイスがクライアントデバイス認証コンポーネントをインストールして実行したときに評価されます。このコンポーネントは、接続情報を使用して有効な CA 証明書を生成し、 PutCertificateAuthoriesオペレーションで AWS IoT Greengrass クラウドサービスにアップロードします。クライアントデバイスは、これらの CA 証明書を使用して、コアデバイスの ID を検証します。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

また、この操作を AWS IoT Greengrass コントロールプレーンで使用して、コアデバイスの接続情報を表示することもできます。詳細については、AWS IoT Greengrass V1 API リファレンスの「GetConnectivityInfo」を参照してください。

リソースタイプ: thing (コアデバイス)

リソース ARN 形式: arn:aws:iot:*region*:*account-id*:thing/*core-device-thing-name*

`greengrass:UpdateConnectivityInfo`

コアデバイスの接続情報を更新するためのアクセス許可を付与します。この情報は、クライアントデバイスがコアデバイスに接続する方法を説明するものです。

このアクセス許可は、コアデバイスがIP 検出コンポーネントを実行したときに評価されます。このコンポーネントは、クライアントデバイスがローカルネットワーク上のコアデバイスに接続するために必要な情報を特定します。その後、このコンポーネントはこの操作を使用して、接続情報を AWS IoT Greengrass クラウドサービスにアップロードし、クライアントデバイスが Discover 操作でこの情報を取得できるようにします。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

また、この操作を AWS IoT Greengrass コントロールプレーンで使用して、コアデバイスの接続情報を手動で更新することもできます。詳細については、AWS IoT Greengrass V1 API リファレンスの「UpdateConnectivityInfo」を参照してください。

リソースタイプ: `thing` (コアデバイス)

リソース ARN 形式: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`core-device-thing-name`*

クライアントデバイスのアクション

`greengrass:Discover`

クライアントデバイスが接続できるコアデバイスの接続情報を検出するためのアクセス許可を付与します。この情報は、クライアントデバイスがコアデバイスに接続する方法を説明するものです。クライアントデバイスは、 BatchAssociateClientDeviceWithCoreDeviceオペレーションを使用して、関連付けたコアデバイスのみを検出できます。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

リソースタイプ: `thing` (クライアントデバイス)

リソース ARN 形式: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`client-device-thing-name`*

コアデバイスの AWS IoT ポリシーを更新する

AWS IoT Greengrass および AWS IoT コンソール、または AWS IoT API を使用して、コアデバイスの AWS IoT ポリシーを表示したり更新したりすることができます。

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアインストーラを使用してリソースをプロビジョニングした場合、コアデバイスには、すべての AWS IoT Greengrass アクション (`greengrass:*`) へのアクセスを許可する AWS IoT ポリシーが存在します。コアデバイスが使用するアクションのみにアクセスを制限するには、次の手順を実行します。

コアデバイスの AWS IoT ポリシーを確認し更新する (コンソール)

1. AWS IoT Greengrass コンソールのナビゲーションメニューで、[Core devices] (コアデバイス) を選択します。
2. [Core devices] (コアデバイス) ページで、更新するコアデバイスを選択します。
3. コアデバイスの詳細ページで、コアデバイスの [Thing] (モノ) へのリンクを選択します。このリンクをクリックすると、AWS IoT コンソールの [thing details] (モノ詳細) ページが開きます。
4. [thing details] (モノ詳細) ページで、[Certificates] (証明書) を選択します。
5. [Certificates] (証明書) タブで、モノのアクティブな証明書を選択します。
6. [certificate details] (証明書詳細) ページで、[Policies] (ポリシー) を選択します。
7. [Policies] (ポリシー) タブで、確認して更新する AWS IoT ポリシーを選択します。コアデバイスのアクティブな証明書にアタッチされている任意のポリシーに、必要なアクセス許可を追加できます。

   > **Note**
   >
   > リソースのプロビジョニングに AWS IoT Greengrass Core ソフトウェアインストーラを使用している場合、2 つの AWS IoT ポリシーが存在します。存在する場合は、GreengrassV2IoTThingPolicy という名前のポリシーを選択することをお勧めします。クイックインストーラで作成するコアデバイスは、デフォルトでこのポリシー名を使用します。このポリシーにアクセス許可を追加すると、このポリシーを使用する他のコアデバイスにもこれらのアクセス許可が付与されます。

8. [policy overview] (ポリシーの概要) で、[Edit active version] (アクティブなバージョンの編集) を選択します。
9. ポリシーを確認し、必要に応じてアクセス許可を追加、削除、または編集します。
10. 新しいポリシーバージョンをアクティブなバージョンとして設定するには、[Policy version status] (ポリシーバージョンのステータス) で、[Set the edited version as the active version for

this policy] (編集したバージョンをこのポリシーのアクティブバージョンとして設定) を選択します。

11. [Save as new version] (新しいバージョンとして保存) を選択します。

コアデバイスの AWS IoT ポリシーを確認し更新する (AWS CLI)

1. コアデバイスの AWS IoT モノのプリンシパルをリスト表示します。モノのプリンシパルは X.509 デバイス証明書またはその他の識別子にすることができます。次のコマンドを実行し、をコアデバイスの名前*MyGreengrassCore*に置き換えます。

```
aws iot list-thing-principals --thing-name MyGreengrassCore
```

この動作は、コアデバイスのモノのプリンシパルをリスト表示するレスポンスを返します。

```
{
    "principals": [
        "arn:aws:iot:us-west-2:123456789012:cert/certificateId"
    ]
}
```

2. コアデバイスのアクティブな証明書を特定します。以下のコマンドを実行して、アクティブな証明書が見つかるまで、*certificateId* を前の手順からの各証明書の ID に置き換えます。証明書 ID は、証明書 ARN の末尾にある 16 進数の文字列です。`--query` 引数は、証明書のステータスのみを出力するように指定しています。

```
aws iot describe-certificate --certificate-id certificateId --query
 'certificateDescription.status'
```

この操作では、証明書の状態が文字列で返されます。例えば、証明書がアクティブな場合、この操作は `"ACTIVE"` を出力します。

3. 証明書にアタッチされている AWS IoT ポリシーをリスト表示します。次のコマンドを実行し、証明書 ARN を証明書の ARN に置き換えます。

```
aws iot list-principal-policies --principal arn:aws:iot:us-
west-2:123456789012:cert/certificateId
```

この操作は、証明書にアタッチされている AWS IoT ポリシーのリストを返します。

```
{
    "policies": [
        {
            "policyName":
 "GreengrassTESCertificatePolicyMyGreengrassCoreTokenExchangeRoleAlias",
            "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassTESCertificatePolicyMyGreengrassCoreTokenExchangeRoleAlias"
        },
        {
            "policyName": "GreengrassV2IoTThingPolicy",
            "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy"
        }
    ]
}
```

4. 表示して更新するポリシーを選択します。

> **Note**
>
> リソースのプロビジョニングに AWS IoT Greengrass Core ソフトウェアインストーラを使用している場合、2 つの AWS IoT ポリシーが存在します。存在する場合は、GreengrassV2IoTThingPolicy という名前のポリシーを選択することをお勧めします。クイックインストーラで作成するコアデバイスは、デフォルトでこのポリシー名を使用します。このポリシーにアクセス許可を追加すると、このポリシーを使用する他のコアデバイスにもこれらのアクセス許可が付与されます。

5. ポリシーのドキュメントを取得します。次のコマンドを実行し、*GreengrassV2IoTThingPolicy* をポリシーの名前に置き換えます。

```
aws iot get-policy --policy-name GreengrassV2IoTThingPolicy
```

この操作は、ポリシーのドキュメントとポリシーに関するその他の情報が含まれるレスポンスを返します。ポリシードキュメントは、文字列としてシリアル化された JSON オブジェクトです。

```
{
    "policyName": "GreengrassV2IoTThingPolicy",
```

```
    "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy",
    "policyDocument": "{\
  \\"Version\\": \\"2012-10-17\\",\
  \\"Statement\\": [\
    {\
      \\"Effect\\": \\"Allow\\",\
      \\"Action\\": [\
                \\"iot:Connect\\",\
                \\"iot:Publish\\",\
                \\"iot:Subscribe\\",\
                \\"iot:Receive\\",\
                \\"greengrass:*\\"\
],\
      \\"Resource\\": \\"*\\"\
    }\
  ]\
}",
    "defaultVersionId": "1",
    "creationDate": "2021-02-05T16:03:14.098000-08:00",
    "lastModifiedDate": "2021-02-05T16:03:14.098000-08:00",
    "generationId":
 "f19144b798534f52c619d44f771a354f1b957dfa2b850625d9f1d0fde530e75f"
}
```

6. オンラインコンバータまたはその他のツールを使用して、ポリシードキュメント文字列を JSON オブジェクトに変換し、`iot-policy.json` という名前のファイルに保存します。

   例えば、jq ツールがインストールされている場合には、次のコマンドを実行してポリシードキュメントを取得し、JSON オブジェクトに変換してから、ポリシードキュメントを JSON オブジェクトとして保存することができます。

   ```
   aws iot get-policy --policy-name GreengrassV2IoTThingPolicy --query
    'policyDocument' | jq fromjson >> iot-policy.json
   ```

7. ポリシードキュメントを確認し、必要に応じてアクセス許可を追加、削除、または編集します。

   例えば、Linux ベースのシステムでは、次のコマンドを実行し、GNU nano を使用してファイルを開きます。

   ```
   nano iot-policy.json
   ```

完成したポリシー文書は、コアデバイス向けの最低限の AWS IoT ポリシーと類似したものになる可能性があります。

8. 変更をポリシーの新しいバージョンとして保存します。次のコマンドを実行し、*GreengrassV2IoTThingPolicy* をポリシーの名前に置き換えます。

```
aws iot create-policy-version --policy-name GreengrassV2IoTThingPolicy --policy-
document file://iot-policy.json --set-as-default
```

成功すると、次の例に類似したレスポンスが返されます。

```
{
    "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy",
    "policyDocument": "{\
  \\"Version\\": \\"2012-10-17\\",\
  \\"Statement\\": [\
    {\
      \\"Effect\\": \\"Allow\\",\
      \\"Action\\": [\
\\t\\t\\"iot:Connect\\",\
\\t\\t\\"iot:Publish\\",\
\\t\\t\\"iot:Subscribe\\",\
\\t\\t\\"iot:Receive\\",\
\\t\\t\\"greengrass:*\\"\
      ],\
      \\"Resource\\": \\"*\\"\
    }\
  ]\
}",
    "policyVersionId": "2",
    "isDefaultVersion": true
}
```

AWS IoT Greengrass V2 コアデバイス向けの最低限の AWS IoT ポリシー

> **Important**
>
> Greengrass nucleus コンポーネントの新しいバージョンでは、最低限の AWS IoT ポリシーに対して追加のアクセス許可が必要となります。追加の許可を付与するにあたり、コアデバイスの AWS IoT ポリシーを更新する必要が生じる場合があります。
>
> - Greengrass nucleus v2.5.0 以降を実行しているコアデバイスでは、モノグループからコアデバイスを削除するときにコンポーネントをアンインストールするにあたり、`greengrass:ListThingGroupsForCoreDevice` アクセス許可を使用します。
> - Greengrass nucleus v2.3.0 以降を実行しているコアデバイスでは、大規模なデプロイ設定ドキュメントをサポートするにあたり、`greengrass:GetDeploymentConfiguration` アクセス許可を使用します。

次のポリシーの例には、コアデバイスの基本的な Greengrass 機能をサポートするのに必要な最小限のアクションが含まれています。

- `Connect` ポリシーには、コアデバイスのモノ名の後に * ワイルドカードが含まれます (例: *core-device-thing-name**)。コアデバイスは、同じデバイス証明書を使用して、AWS IoT Core への複数の同時サブスクリプションを実行しようと試みますが、接続のクライアント ID がコアデバイスのモノ名と完全に一致しない場合があります。最初の 50 件のサブスクリプション以降、コアデバイスは *core-device-thing-name*#*number* をクライアント ID として使用します。ここにある *number* は、サブスクリプションが 50 件増えるごと増分されます。例えば、`MyCoreDevice` という名のコアデバイスが 150 件の同時サブスクリプションを作成する場合には、次のクライアント ID を使用します。
  - サブスクリプション 1 から 50: `MyCoreDevice`
  - サブスクリプション 51 から 100: `MyCoreDevice#2`
  - サブスクリプション 101 から 150: `MyCoreDevice#3`

  ワイルドカードを使用すると、サフィックスのあるクライアント ID を使用する場合に、コアデバイスが接続できるようになります。
- ポリシーには、シャドウステータスに使用されるトピックを含む、コアデバイスがメッセージを発行、サブスクライブし、メッセージを受信できる、MQTT トピックとトピックのフィルターが一覧表示されます。AWS IoT Core、Greengrass コンポーネント、およびクライアントデバイス間のメッセージ交換に対応するためには、許可するトピックとトピックのフィルターを指定します。詳

細については、「AWS IoT Core デベロッパーガイド」の「パブリッシュ/サブスクライブポリシーの例」を参照してください。

- このポリシーは、テレメトリデータに関する次のトピックにパブリッシュするためのアクセス許可を付与します。

  ```
  $aws/things/core-device-thing-name/greengrass/health/json
  ```

  テレメトリを無効にしたコアデバイスでは、このアクセス許可を外すことができます。詳細については、「AWS IoT Greengrass コアデバイスからシステムヘルステレメトリデータを収集する」を参照してください。

- このポリシーは、AWS IoT ロールエイリアスを介して IAM ロールを継承するためのアクセス許可を付与します。コアデバイスは、トークン交換ロールと呼ばれるこのロールを使用して、AWS リクエストの認証に使用することができる AWS 認証情報を取得します。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

  AWS IoT Greengrass Core ソフトウェアをインストールするときに、このアクセス許可のみが含まれる 2 つ目の AWS IoT ポリシーを作成してアタッチします。コアデバイスのプライマリ AWS IoT ポリシーにこのアクセス許可を含める場合には、他の AWS IoT ポリシーを外して削除することができます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "arn:aws:iot:region:account-id:client/core-device-thing-name*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive",
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:topic/$aws/things/core-device-thing-
name/greengrass/health/json",
```

```
                "arn:aws:iot:region:account-id:topic/$aws/things/core-device-thing-
name/greengrassv2/health/json",
                "arn:aws:iot:region:account-id:topic/$aws/things/core-device-thing-
name/jobs/*",
                "arn:aws:iot:region:account-id:topic/$aws/things/core-device-thing-
name/shadow/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:topicfilter/$aws/things/core-device-
thing-name/jobs/*",
                "arn:aws:iot:region:account-id:topicfilter/$aws/things/core-device-
thing-name/shadow/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iot:AssumeRoleWithCertificate",
            "Resource": "arn:aws:iot:region:account-id:rolealias/token-exchange-role-
alias-name"
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:GetComponentVersionArtifact",
                "greengrass:ResolveComponentCandidates",
                "greengrass:GetDeploymentConfiguration",
                "greengrass:ListThingGroupsForCoreDevice"
            ],
            "Resource": "*"
        }
    ]
}
```

クライアントデバイスをサポートするための最低限の AWS IoT ポリシー

次のポリシー例には、コアデバイス上のクライアントデバイスとの対話をサポートするために必要となる最低限のアクションが含まれています。クライアントデバイスをサポートするため、コアデバイ

スは基本操作のための最低限の AWS IoT ポリシーに加えて、この AWS IoT ポリシーでもアクセス許可を持つ必要があります。

- このポリシーにより、コアデバイスは自身の接続情報を更新できるようになります。このアクセス許可 (greengrass:UpdateConnectivityInfo) は、コアデバイスに IP 検出コンポーネントをデプロイする場合にのみ必要となります。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:topic/$aws/things/core-device-thing-
name-gci/shadow/get"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:topicfilter/$aws/things/core-device-
thing-name-gci/shadow/update/delta",
                "arn:aws:iot:region:account-id:topicfilter/$aws/things/core-device-
thing-name-gci/shadow/get/accepted"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:topic/$aws/things/core-device-thing-
name-gci/shadow/update/delta",
                "arn:aws:iot:region:account-id:topic/$aws/things/core-device-thing-
name-gci/shadow/get/accepted"
```

```
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:PutCertificateAuthorities",
                "greengrass:VerifyClientDeviceIdentity"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:VerifyClientDeviceIoTCertificateAssociation"
            ],
            "Resource": "arn:aws:iot:region:account-id:thing/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:GetConnectivityInfo",
                "greengrass:UpdateConnectivityInfo"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:thing/core-device-thing-name"
            ]
        }
    ]
}
```

クライアントデバイス向けの最低限の AWS IoT ポリシー

次のポリシー例には、クライアントデバイスが MQTT 経由で接続して通信するコアデバイスを検出するために必要となる最低限のアクションが含まれています。クライアントデバイスの AWS IoT ポリシーには、デバイスが関連する Greengrass コアデバイスの接続情報を検出できるようにするための `greengrass:Discover` アクションを含める必要があります。`Resource` セクションには、Greengrass コアデバイスの ARN ではなく、クライアントデバイスの Amazon リソースネーム (ARN) を指定します。

- このポリシーは、すべての MQTT トピックでの通信を許可します。セキュリティのベストプラクティスに沿うように、`iot:Publish`、`iot:Subscribe`、および `iot:Receive` のアクセス許可は、ユースケースに必要とされる最小限のトピックセットのみに制限してください。

- このポリシーにより、モノはあらゆる AWS IoT モノのコアデバイスを検出できるようになります。セキュリティのベストプラクティスに沿うように、greengrass:Discover のアクセス許可はクライアントデバイスの AWS IoT モノまたは AWS IoT モノのセットに一致するワイルドカードのみに制限してください。

> **Important**
>
> モノのポリシー変数 (iot:Connection.Thing.*) は コアデバイスまたは Greengrass データプレーン操作用の AWS IoT ポリシーではサポートされていません。代わりに、ワイルドカードを使用して名前が似ている複数のデバイスと一致させることができます。たとえば、MyGreengrassDevice* と指定すると MyGreengrassDevice1、MyGreengrassDevice2 などと一致します。

- クライアントデバイスの AWS IoT ポリシーでは通常、iot:GetThingShadow、iot:UpdateThingShadow、または iot:DeleteThingShadow アクションに対するアクセス許可は必要ありません。これは、クライアントデバイスのシャドウ同期操作は Greengrass コアデバイスで処理されるからです。コアデバイスがクライアントのデバイスシャドウを処理できるようにするには、コアデバイスの AWS IoT ポリシーでこれらのアクションが許可されていることと、Resource セクションにクライアントデバイスの ARN が含まれていることを確認します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:topic/*"
            ]
```

```
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:topicfilter/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:topic/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:Discover"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:thing/*"
            ]
        }
    ]
}
```

AWS IoT Greengrass のためのアイデンティティおよびアクセス管理

AWS Identity and Access Management (IAM) は、管理者が AWS リソースへのアクセスを安全に制御するために役立つ AWS のサービス です。IAM 管理者は、誰をわ認証 (サインイン) し、誰に AWS IoT Greengrass リソースの使用を許可する (権限を持たせる) かを制御します。IAM は、無料で使用できる AWS のサービスです。

> **Note**
>
> このトピックでは、IAM の概念と機能について説明します。AWS IoT Greengrass でサポートされる IAM の機能の詳細については、「the section called “AWS IoT Greengrass と IAM の連携について”」を参照してください。

対象者

AWS Identity and Access Management (IAM) の用途は、AWS IoT Greengrass で行う作業によって異なります。

サービスユーザー - AWS IoT Greengrass サービスを使用してジョブを実行する場合は、必要な権限と認証情報を管理者が用意します。作業を実行するためにさらに多くの AWS IoT Greengrass 機能を使用するとき、追加の権限が必要になる場合があります。アクセスの管理方法を理解すると、管理者から適切な権限をリクエストするのに役に立ちます。AWS IoT Greengrass 機能にアクセスできない場合は、「AWS IoT Greengrass のアイデンティティとアクセスの問題のトラブルシューティング」を参照してください。

サービス管理者 - 社内の AWS IoT Greengrass リソースを担当している場合は、通常、AWS IoT Greengrass への完全なアクセスがあります。サービスのユーザーがどの AWS IoT Greengrass 機能やリソースにアクセスするかを決めるのは管理者の仕事です。その後、IAM 管理者にリクエストを送信して、サービスユーザーの権限を変更する必要があります。このページの情報を確認して、IAM の基本概念を理解してください。お客様の会社で AWS IoT Greengrass で IAM を利用する方法の詳細については、「AWS IoT Greengrass と IAM の連携について」を参照してください。

IAM 管理者 - 管理者は、AWS IoT Greengrass へのアクセスを管理するポリシーの書き込み方法の詳細について確認する場合があります。IAM で使用できる AWS IoT Greengrass アイデンティティベースのポリシーの例を表示するには、「AWS IoT Greengrass のアイデンティティベースのポリシーの例」を参照してください。

アイデンティティを使用した認証

認証とは、アイデンティティ認証情報を使用して AWSにサインインする方法です。ユーザーは、AWS アカウントのルートユーザーもしくは IAM ユーザーとして、または IAM ロールを引き受けることによって、認証を受ける (AWS にサインインする) 必要があります。

ID ソースから提供された認証情報を使用して、フェデレーティッドアイデンティティとして AWS にサインインできます。 AWS IAM Identity Centerフェデレーティッドアイデンティティの例として

は、(IAM Identity Center) ユーザー、会社のシングルサインオン認証、Google または Facebook の認証情報などがあります。フェデレーティッドアイデンティティとしてサインインする場合、IAM ロールを使用して、前もって管理者により ID フェデレーションが設定されています。フェデレーションを使用して AWSにアクセスする場合、間接的にロールを引き受けることになります。

ユーザーのタイプに応じて、AWS Management Console または AWS アクセスポータルにサインインできます。AWS へのサインインの詳細については、『AWS サインイン ユーザーガイド』の「AWS アカウント にサインインする方法」を参照してください。

プログラムで AWS にアクセスする場合、AWS は Software Development Kit (SDK) とコマンドラインインターフェイス (CLI) を提供し、認証情報でリクエストに暗号で署名します。AWS ツールを使用しない場合は、リクエストに自分で署名する必要があります。リクエストに署名する推奨方法の使用については、『IAM ユーザーガイド』の「AWS API リクエストの署名」を参照してください。

使用する認証方法を問わず、追加のセキュリティ情報の提供が求められる場合もあります。例えば、AWS では、アカウントのセキュリティ強化のために多要素認証 (MFA) の使用をお勧めしています。詳細については、「AWS IAM Identity Center ユーザーガイド」の「Multi-factor authentication (多要素認証)」 および「IAM ユーザーガイド」の「AWS での多要素認証 (MFA) の使用」を参照してください。

AWS アカウントのルートユーザー

AWS アカウントを作成する場合は、そのアカウントのすべての AWS のサービスとリソースに対して完全なアクセス権を持つ 1 つのサインインアイデンティティから始めます。このアイデンティティは AWS アカウントのルートユーザーと呼ばれ、アカウントの作成に使用した E メールアドレスとパスワードでサインインすることによってアクセスできます。日常的なタスクには、ルートユーザーを使用しないことを強くお勧めします。ルートユーザーの認証情報は保護し、ルートユーザーでしか実行できないタスクを実行するときに使用します。ルートユーザーとしてサインインする必要があるタスクの完全なリストについては、『IAM ユーザーガイド』の「ルートユーザー認証情報が必要なタスク」を参照してください。

IAM ユーザーとグループ

IAM ユーザーは、1 人のユーザーまたは 1 つのアプリケーションに対して特定の権限を持つ AWS アカウント 内のアイデンティティです。可能であれば、パスワードやアクセスキーなどの長期的な認証情報を保有する IAM ユーザーを作成する代わりに、一時的な認証情報を使用することをお勧めします。ただし、IAM ユーザーでの長期的な認証情報が必要な特定のユースケースがある場合は、アクセスキーをローテーションすることをお勧めします。詳細については、「IAM ユーザーガイド」

の「長期的な認証情報を必要とするユースケースのためにアクセスキーを定期的にローテーションする」を参照してください。

IAM グループは、IAM ユーザーの集団を指定するアイデンティティです。グループとしてサインインすることはできません。グループを使用して、複数のユーザーに対して一度に権限を指定できます。多数のユーザーグループがある場合、グループを使用することで権限の管理が容易になります。例えば、IAMAdmins という名前のグループを設定して、そのグループに IAM リソースを管理する権限を与えることができます。

ユーザーは、ロールとは異なります。ユーザーは 1 人の人または 1 つのアプリケーションに一意に関連付けられますが、ロールはそれを必要とする任意の人が引き受けるようになっています。ユーザーには永続的な長期の認証情報がありますが、ロールでは一時的な認証情報が提供されます。詳細については、「IAM ユーザーガイド」の「IAM ユーザー (ロールではなく) の作成が適している場合」を参照してください。

IAM ロール

IAM ロールは、特定の権限を持つ、AWS アカウント 内のアイデンティティです。これは IAM ユーザーに似ていますが、特定のユーザーには関連付けられていません。ロールを切り替えることによって、AWS Management Console で IAM ロールを一時的に引き受けることができます。ロールを引き受けるには、AWS CLI または AWS API オペレーションを呼び出すか、カスタム URL を使用します。ロールを使用する方法の詳細については、「IAM ユーザーガイド」の「IAM ロールの使用」を参照してください。

一時的な認証情報を持った IAM ロールは、以下の状況で役立ちます。

- フェデレーションユーザーユーザーアクセス – フェデレーションアイデンティティに権限を割り当てるには、ロールを作成してそのロールの権限を定義します。フェデレーティッドアイデンティティが認証されると、そのアイデンティティはロールに関連付けられ、ロールで定義されている権限が付与されます。フェデレーションの詳細については、「IAM ユーザーガイド」の「サードパーティー ID プロバイダー向けロールの作成」を参照してください。IAM アイデンティティセンターを使用する場合、権限セットを設定します。アイデンティティが認証後にアクセスできるものを制御するため、IAM Identity Center は、権限セットを IAM のロールに関連付けます。権限セットの詳細については、「AWS IAM Identity Center ユーザーガイド」の「権限セット」を参照してください。
- 一時的な IAM ユーザー権限 - IAM ユーザーまたはロールは、特定のタスクに対して複数の異なる権限を一時的に IAM ロールで引き受けることができます。
- クロスアカウントアクセス - IAM ロールを使用して、自分のアカウントのリソースにアクセスすることを、別のアカウントの人物 (信頼済みプリンシパル) に許可できます。クロスアカウント

アクセス権を付与する主な方法は、ロールを使用することです。ただし、一部の AWS のサービスでは、(ロールをプロキシとして使用する代わりに) リソースにポリシーを直接アタッチできます。クロスアカウントアクセスにおけるロールとリソースベースのポリシーの違いについては、「IAM ユーザーガイド」の「IAM ロールとリソースベースのポリシーとの相違点」を参照してください。

- クロスサービスアクセス - 一部の AWS のサービスでは、他の AWS のサービスの機能を使用します。例えば、あるサービスで呼び出しを行うと、通常そのサービスによって Amazon EC2 でアプリケーションが実行されたり、Amazon S3 にオブジェクトが保存されたりします。サービスでは、呼び出し元プリンシパルの権限、サービスロール、またはサービスリンクロールを使用してこれを行う場合があります。
  - 転送アクセスセッション (FAS) – IAM ユーザーまたはロールを使用して AWS でアクションを実行するユーザーは、プリンシパルと見なされます。一部のサービスを使用する際に、アクションを実行することで、別のサービスの別のアクションがトリガーされることがあります。FAS は、AWS のサービスを呼び出すプリンシパルの権限を、AWS のサービスのリクエストと合わせて使用し、ダウンストリームのサービスに対してリクエストを行います。FAS リクエストは、サービスが、完了するために他の AWS のサービス またはリソースとのやりとりを必要とするリクエストを受け取ったときにのみ行われます。この場合、両方のアクションを実行するためのアクセス許可が必要です。FAS リクエストを行う際のポリシーの詳細については、「転送アクセスセッション」を参照してください。
  - サービスロール - サービスがユーザーに代わってアクションを実行するために引き受ける IAM ロールです。IAM 管理者は、IAM 内からサービスロールを作成、変更、削除できます。詳細については、「IAM ユーザーガイド」の「AWS のサービスにアクセス許可を委任するロールの作成」を参照してください。
  - サービスリンクロール - サービスリンクロールは、AWS のサービス にリンクされたサービスロールの一種です。サービスがロールを引き受け、ユーザーに代わってアクションを実行できるようになります。サービスリンクロールは、AWS アカウント に表示され、サービスによって所有されます。IAM 管理者は、サービスリンクロールの権限を表示できますが、編集することはできません。
- Amazon EC2 で実行されているアプリケーション - EC2 インスタンスで実行され、AWS CLI または AWS API 要求を行っているアプリケーションの一時的な認証情報を管理するには、IAM ロールを使用できます。これは、EC2 インスタンス内でのアクセスキーの保存に推奨されます。AWS ロールを EC2 インスタンスに割り当て、そのすべてのアプリケーションで使用できるようにするには、インスタンスに添付されたインスタンスプロファイルを作成します。インスタンスプロファイルにはロールが含まれ、EC2 インスタンスで実行されるプログラムは一時的な認証情報を取得

できます。詳細については、「IAM ユーザーガイド」の「Amazon EC2 インスタンスで実行されるアプリケーションに IAM ロールを使用してアクセス許可を付与する」を参照してください。

IAM ロールと IAM ユーザーのどちらを使用するかについては、「IAM ユーザーガイド」の「(IAM ユーザーではなく) IAM ロールをいつ作成したら良いのか?」を参照してください。

ポリシーを使用したアクセス権の管理

AWS でアクセス権を管理するには、ポリシーを作成して AWS アイデンティティまたはリソースにアタッチします。ポリシーは AWS のオブジェクトであり、アイデンティティやリソースに関連付けて、これらの権限を定義します。AWS は、プリンシパル (ユーザー、ルートユーザー、またはロールセッション) がリクエストを行うと、これらのポリシーを評価します。ポリシーでの権限により、リクエストが許可されるか拒否されるかが決まります。大半のポリシーは JSON ドキュメントとして AWSに保存されます。JSON ポリシードキュメントの構造と内容の詳細については、「IAM ユーザーガイド」の「JSON ポリシー概要」を参照してください。

管理者は AWSJSON ポリシーを使用して、だれが何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

デフォルトでは、ユーザーやロールに権限はありません。IAM 管理者は、リソースで必要なアクションを実行するための権限をユーザーに付与する IAM ポリシーを作成できます。その後、管理者はロールに IAM ポリシーを追加し、ユーザーはロールを引き継ぐことができます。

IAM ポリシーは、オペレーションの実行方法を問わず、アクションの権限を定義します。例えば、`iam:GetRole` アクションを許可するポリシーがあるとします。このポリシーがあるユーザーは、AWS Management Console、AWS CLI、または AWS API からロール情報を取得できます。

アイデンティティベースポリシー

アイデンティティベースポリシーは、IAM ユーザー、ユーザーのグループ、ロールなど、アイデンティティにアタッチできる JSON 権限ポリシードキュメントです。これらのポリシーは、ユーザーとロールが実行できるアクション、リソース、および条件を制御します。アイデンティティベースのポリシーを作成する方法については、「IAM ユーザーガイド」の「IAM ポリシーの作成」を参照してください。

アイデンティティベースポリシーは、さらにインラインポリシーまたはマネージドポリシーに分類できます。インラインポリシーは、単一のユーザー、グループ、またはロールに直接埋め込まれます。管理ポリシーは、AWS アカウント 内の複数のユーザー、グループ、およびロールにアタッチできる

スタンドアロンポリシーです。マネージドポリシーには、AWS マネージドポリシーとカスタマー管理ポリシーがあります。マネージドポリシーまたはインラインポリシーのいずれかを選択する方法については、「IAM ユーザーガイド」の「マネージドポリシーとインラインポリシーの比較」を参照してください。

リソースベースのポリシー

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには、例として、IAM ロールの信頼ポリシーや Amazon S3 バケットポリシーがあります。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。ポリシーがアタッチされているリソースの場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーでは、プリンシパルを指定する必要があります。プリンシパルには、アカウント、ユーザー、ロール、フェデレーションユーザー、または AWS のサービスを含めることができます。

リソースベースのポリシーは、そのサービス内にあるインラインポリシーです。リソースベースのポリシーでは IAM の AWSマネージドポリシーは使用できません。

アクセスコントロールリスト (ACL)

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするための権限を持つかをコントロールします。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

Simple Storage Service (Amazon S3)、AWS WAF、および Amazon VPC は、ACL をサポートするサービスの例です。ACL の詳細については、「Amazon Simple Storage Service デベロッパーガイド」の「アクセスコントロールリスト (ACL) の概要」を参照してください。

その他のポリシータイプ

AWS では、他の一般的ではないポリシータイプをサポートしています。これらのポリシータイプでは、より一般的なポリシータイプで付与された最大の権限を設定できます。

- 権限の境界 - 権限の境界は、アイデンティティベースのポリシーによって IAM エンティティ (IAM ユーザーまたはロール) に付与できる許可の上限を設定する高度な機能です。エンティティに権限の境界を設定できます。結果として得られる権限は、エンティティのアイデンティティベースポリシーとその権限の境界の共通部分になります。`Principal` フィールドでユーザーまたはロールを指定するリソースベースのポリシーでは、権限の境界は制限されません。これらのポリシーの

いずれかを明示的に拒否した場合、権限は無効になります。権限の境界の詳細については、「IAM ユーザーガイド」の「IAM エンティティの権限の境界」を参照してください。

- サービスコントロールポリシー (SCP) - SCP は、AWS Organizations で組織や組織単位 (OU) の最大権限を指定する JSON ポリシーです。AWS Organizations は、顧客のビジネスが所有する複数の AWS アカウント をグループ化し、一元的に管理するサービスです。組織内のすべての機能を有効にすると、サービスコントロールポリシー (SCP) を一部またはすべてのアカウントに適用できます。SCP はメンバーアカウントのエンティティに対する権限を制限します (各 AWS アカウントのルートユーザー など)。Organizations と SCP の詳細については、『AWS Organizations ユーザーガイド』の「SCP の仕組み」を参照してください。
- セッションポリシー - セッションポリシーは、ロールまたはフェデレーションユーザーの一時的なセッションをプログラムで作成する際にパラメータとして渡す高度なポリシーです。結果としてセッションの権限の範囲は、ユーザーまたはロールのアイデンティティベースポリシーとセッションポリシーの共通部分になります。また、リソースベースのポリシーから権限が派生する場合もあります。これらのポリシーのいずれかを明示的に拒否した場合、権限は無効になります。詳細については、「IAM ユーザーガイド」の「セッションポリシー」をご参照ください。

複数のポリシータイプ

1 つのリクエストに複数のタイプのポリシーが適用されると、結果として作成される権限を理解するのがさらに難しくなります。複数のポリシータイプが関連するとき、リクエストを許可するかどうかを AWS が決定する方法の詳細については、『IAM ユーザーガイド』の「Policy evaluation logic (ポリシーの評価ロジック)を参照してください。

以下も参照してください。

- the section called “AWS IoT Greengrass と IAM の連携について”
- the section called “アイデンティティベースポリシーの例”
- the section called “アイデンティティとアクセスの問題のトラブルシューティング”

AWS IoT Greengrass と IAM の連携について

IAM を使用して AWS IoT Greengrass へのアクセスを管理するには、AWS IoT Greengrass で使用できる IAM の機能を理解しておく必要があります。

IAM 機能	Greengrass によってサポートされていますか。
リソースレベルのアクセス許可を持つアイデンティティベースポリシー	はい
リソースベースのポリシー	いいえ
アクセスコントロールリスト (ACL)	いいえ
タグベースの承認	はい
一時的な認証情報	はい
サービスにリンクされたロール	いいえ
サービスロール	はい

その他の AWS のサービスが IAM と連携する方法の概要を把握するには、「IAM ユーザーガイド」の「IAM と連携する AWS サービス」を参照してください。

AWS IoT Greengrass のアイデンティティベースポリシー

IAM アイデンティティベースのポリシーでは、許可または拒否されたアクションとリソースを指定でき、さらにアクションが許可または拒否された条件を指定できます。AWS IoT Greengrass は、特定のアクション、リソース、および条件キーをサポートします。ポリシーで使用するすべての要素については、「IAM ユーザーガイド」の「[IAM JSON policy elements reference] (IAM JSON ポリシーエレメントのリファレンス)」を参照してください。

アクション

管理者は AWS JSON ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どの プリンシパル がどの リソース に対してどのような 条件 下で アクション を実行できるかということです。

JSON ポリシーの `Action` 要素には、ポリシー内のアクセスを許可または拒否するために使用できるアクションが記述されます。ポリシーアクションの名前は通常、関連する AWS API オペレーションと同じです。一致する API オペレーションのない許可のみのアクションなど、いくつかの例外が

あります。また、ポリシーに複数のアクションが必要なオペレーションもあります。これらの追加アクションは、依存アクションと呼ばれます。

このアクションは、関連付けられたオペレーションを実行するための許可を付与するポリシーで使用されます。

AWS IoT Greengrass のポリシーアクションは、アクションの前に `greengrass:` プレフィックスを使用します。例えば、`ListCoreDevices` API オペレーションを使用して AWS アカウント のコアデバイスを一覧表示するには、ポリシーに `greengrass:ListCoreDevices` アクションを含めます。ポリシーステートメントには、`Action` 要素または `NotAction` 要素のいずれかを含める必要があります。 AWS IoT Greengrass は、このサービスで実行できるタスクを説明する独自の一連のアクションを定義します。

1 つのステートメントで複数のアクションを指定するには、次のようにアクションをカンマで区切って全体を括弧 ([ ]) で囲って表示します。

```
"Action": [
  "greengrass:action1",
  "greengrass:action2",
  "greengrass:action3"
]
```

ワイルドカード (*) を使用して、複数のアクションを指定できます。たとえば、`List` という単語で始まるすべてのアクションを指定するには、次のアクションを含めます。

```
"Action": "greengrass:List*"
```

> **Note**
>
> サービスに対して使用可能なすべてのアクションを指定するには、ワイルドカードを使用しないことをお勧めします。ベストプラクティスとして、ポリシー内で最小限の特権と狭い範囲のアクセス許可を付与する必要があります。詳細については、「the section called “最小限のアクセス許可の付与”」を参照してください。

AWS IoT Greengrass アクションの詳細な一覧については、「IAM ユーザーガイド」の「AWS IoT Greengrass で定義されるアクション」を参照してください。

リソース

管理者は AWS JSON ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

Resource JSON ポリシー要素は、オブジェクトあるいはアクションが適用されるオブジェクトを指定します。ステートメントには、Resource または NotResource 要素を含める必要があります。ベストプラクティスとして、Amazon リソースネーム (ARN) を使用してリソースを指定します。これは、リソースレベルの許可と呼ばれる特定のリソースタイプをサポートするアクションに対して実行できます。

オペレーションのリスト化など、リソースレベルの許可をサポートしないアクションの場合は、ステートメントがすべてのリソースに適用されることを示すために、ワイルドカード (*) を使用します。

```
"Resource": "*"
```

次の表に、ポリシーステートメントの Resource 要素で使用できる AWS IoT Greengrass リソース ARN を示します。AWS IoT Greengrass アクションでサポートされるリソースレベル権限のマッピングについては、「IAM ユーザーガイド」の「AWS IoT Greengrass で定義されるアクション」を参照してください。

一部の AWS IoT Greengrass アクション (一部のリストオペレーションなど) は、特定のリソースに対して実行できません。このような場合は、ワイルドカードのみを使用する必要があります。

```
"Resource": "*"
```

ステートメントで複数のリソース ARN を指定するには、次のようにアクションをカンマで区切って全体を括弧 ([ ]) で囲って表示します。

```
"Resource": [
  "resource-arn1",
  "resource-arn2",
  "resource-arn3"
]
```

ARN 形式の詳細については、「Amazon Web Services 全般のリファレンス」の「Amazon リソースネーム (ARN) と AWS サービスの名前空間」を参照してください。

条件キー

管理者は AWS JSON ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

Condition 要素 (または Condition ブロック) を使用すると、ステートメントが有効な条件を指定できます。Condition 要素はオプションです。イコールや未満などの条件演算子を使用して条件式を作成することで、ポリシーの条件とリクエスト内の値を一致させることができます。

1 つのステートメントに複数の Condition 要素を指定する場合、または 1 つの Condition 要素に複数のキーを指定する場合、AWS では AND 論理演算子を使用してそれらを評価します。単一の条件キーに複数の値を指定する場合、AWS では OR 論理演算子を使用して条件を評価します。ステートメントの許可が付与される前にすべての条件が満たされる必要があります。

条件を指定する際にプレースホルダー変数も使用できます。例えば IAM ユーザーに、IAM ユーザー名がタグ付けされている場合のみリソースにアクセスできる許可を付与することができます。詳細については、IAM ユーザーガイドの「IAM ポリシーの要素: 変数およびタグ」を参照してください。

AWS はグローバル条件キーとサービス固有の条件キーをサポートしています。すべての AWS グローバル条件キーを確認するには、IAM ユーザーガイドの「AWS グローバル条件コンテキストキー」を参照してください。

例

AWS IoT Greengrass アイデンティティベースのポリシーの例を表示するには、the section called “アイデンティティベースポリシーの例”を参照してください。

AWS IoT Greengrass のリソースベースのポリシー

AWS IoT Greengrass では、リソースベースのポリシーはサポートされていません。

アクセスコントロールリスト (ACL)

AWS IoT Greengrass では ACL はサポートされません。

AWS IoT Greengrass タグに基づく認可

タグをサポートされている AWS IoT Greengrass リソースにアタッチするか、AWS IoT Greengrass へのリクエストでタグを渡すことができます。タグに基づいてアクセスを管理するには、aws:ResourceTag/${TagKey}、aws:RequestTag/${TagKey}、または aws:TagKeys の

条件キーを使用して、ポリシーの [Condition element] (条件要素) でタグ情報を提供します。詳細については、「リソースのタグ付け」を参照してください。

AWS IoT Greengrass の IAM ロール

IAM ロール は、特定のアクセス許可を持つ、AWS アカウント 内のエンティティです。

AWS IoT Greengrass での一時的な認証情報の使用

一時的な認証情報は、フェデレーションでサインイン、IAM ロールを引き受ける、またはクロスアカウントロールを引き受けるために使用されます。一時的なセキュリティ認証情報を取得するには、AssumeRole または GetFederationToken などの AWS STS API オペレーションを呼び出します。

Greengrass コアでは、[device role] (デバイスロール) の一時的な認証情報が Greengrass コンポーネントで利用可能になります。コンポーネントが AWS SDK を使用している場合、AWS SDK がこれを行うため、認証情報を取得するためのロジックを追加する必要はありません。

サービスにリンクされたロール

AWS IoT Greengrass では、サービスにリンクされたロールがサポートされていません。

サービスロール

この機能により、ユーザーに代わってサービスがサービスロールを引き受けることが許可されます。このロールにより、サービスがユーザーに代わって他のサービスのリソースにアクセスし、アクションを完了することが許可されます。サービスロールは、IAM アカウントに表示され、アカウントによって所有されます。つまり、IAM 管理者が、このロールの許可を変更することができます。ただし、これを行うことにより、サービスの機能が損なわれる場合があります。

AWS IoT Greengrass コアデバイスはサービスロールを使用して、Greengrass コンポーネントと Lambda 関数がユーザーに代わって AWS リソースの一部にアクセスできるようにします。詳細については、「the section called “コアデバイスが AWS サービスを操作できるように認証する”」を参照してください。

AWS IoT Greengrass は、サービスロールを使用して、ユーザーに代わって AWS リソースの一部にアクセスします。詳細については、「Greengrass サービスロール」を参照してください。

AWS IoT Greengrass のアイデンティティベースのポリシーの例

デフォルトでは、IAM ユーザーおよびロールには、AWS IoT Greengrass リソースを作成または変更するアクセス許可はありません。また、AWS Management Console や AWS CLI、AWS API を使用

してタスクを実行することもできません。IAM 管理者は、ユーザーとロールに必要な、指定されたリソースで特定の API オペレーションを実行する許可をユーザーとロールに付与する IAM ポリシーを作成する必要があります。続いて、管理者はそれらの許可が必要な IAM ユーザーまたはグループにそのポリシーを添付します。

ポリシーのベストプラクティス

ID ベースのポリシーは、ユーザーのアカウントで誰かが AWS IoT Greengrass リソースを作成、アクセス、または削除できるどうかを決定します。これらのアクションを実行すると、AWS アカウント に追加料金が発生する可能性があります。アイデンティティベースポリシーを作成したり編集したりする際には、以下のガイドラインと推奨事項に従ってください。

- AWS マネージドポリシーを使用して開始し、最小特権の許可に移行する – ユーザーとワークロードへの許可の付与を開始するには、多くの一般的なユースケースのために許可を付与する AWS マネージドポリシーを使用します。これらは AWS アカウント で使用できます。ユースケースに応じた AWS カスタマーマネージドポリシーを定義することで、許可をさらに減らすことをお勧めします。詳細については、「IAM ユーザーガイド」の「AWS マネージドポリシー」または「AWS ジョブ機能の管理ポリシー」を参照してください。
- 最小特権を適用する – IAM ポリシーで許可を設定するときは、タスクの実行に必要な許可のみを付与します。これを行うには、特定の条件下で特定のリソースに対して実行できるアクションを定義します。これは、最小特権アクセス許可とも呼ばれています。IAM を使用して許可を適用する方法の詳細については、「IAM ユーザーガイド」の「IAM でのポリシーとアクセス許可」を参照してください。
- IAM ポリシーで条件を使用してアクセスをさらに制限する – ポリシーに条件を追加して、アクションやリソースへのアクセスを制限できます。例えば、ポリシー条件を記述して、すべてのリクエストを SSL を使用して送信するように指定することができます。また、AWS のサービス などの特定の AWS CloudFormation を介して使用する場合、条件を使用してサービスアクションへのアクセスを許可することもできます。詳細については、[IAM User Guide] (IAM ユーザーガイド) の [IAM JSON policy elements: Condition] (IAM JSON ポリシー要素：条件) を参照してください。
- IAM Access Analyzer を使用して IAM ポリシーを検証し、安全で機能的な許可を確保する - IAM Access Analyzer は、新規および既存のポリシーを検証して、ポリシーが IAM ポリシー言語 (JSON) および IAM のベストプラクティスに準拠するようにします。IAM Access Analyzer は 100 を超えるポリシーチェックと実用的な推奨事項を提供し、安全で機能的なポリシーを作成できるようサポートします。詳細については、「IAM ユーザーガイド」の「IAM Access Analyzer ポリシーの検証」を参照してください。
- 多要素認証 (MFA) を必須にする – AWS アカウントの IAM ユーザーまたはルートユーザーが必要となるシナリオがある場合は、セキュリティを強化するために MFA を有効にします。API オペ

レーションが呼び出されるときに MFA を必須にするには、ポリシーに MFA 条件を追加します。詳細については、「IAM ユーザーガイド」の「MFA 保護 API アクセスの設定」を参照してください。

IAM でのベストプラクティスの詳細については、「IAM ユーザーガイド」の「IAM でのセキュリティのベストプラクティス」を参照してください。

ポリシーの例

以下のお客様定義のポリシーは、一般的なシナリオのアクセス許可を付与します。

例

- ユーザーが自分の許可を表示できるようにする

これらの JSON ポリシードキュメント例を使用して IAM のアイデンティティベースのポリシーを作成する方法については、IAM ユーザーガイドの「JSON タブでのポリシーの作成」を参照してください。

ユーザーが自分の許可を表示できるようにする

この例では、ユーザーアイデンティティに添付されたインラインおよびマネージドポリシーの表示を IAM ユーザーに許可するポリシーを作成する方法を示します。このポリシーには、コンソールで、または AWS CLI か AWS API を使用してプログラム的に、このアクションを完了するアクセス許可が含まれています。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
```

```
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

コアデバイスが AWS サービスを操作できるように認証する

AWS IoT Greengrass コアデバイスは、AWS IoT Core 認証情報プロバイダを使用して AWS サービスへの呼び出しを認証します。AWS IoT Core 認証情報プロバイダは、AWS リクエストを認証するためのユニークなデバイス ID として、デバイスが X.509 証明書を使用できるようにします。これにより、AWS IoT Greengrass コアデバイスに AWS アクセスキー ID とシークレットアクセスキーを保存する必要がなくなります。詳細については、「AWS IoT Core デベロッパーガイド」の「AWS サービスへの直接呼び出しを認証する」を参照してください。

AWS IoT Greengrass コアソフトウェアを実行する際に、コアデバイスが必要とする AWS リソースのプロビジョニングを選択することができます。これには、AWS IoT Core 認証情報プロバイダを通じてコアデバイスが継承する AWS Identity and Access Management (IAM) ロールも含まれます。`--provision true` 引数を使用して、コアデバイスが一時的な AWS 認証情報を取得するための役割とポリシーを設定します。この引数は、この IAM ロールを指し示す AWS IoT ロールエイリアスも設定します。使用する IAM ロール名と AWS IoT ロールエイリアスは、ユーザーが指定することができます。これらの他の名前パラメータなしで `--provision true` を指定した場合、Greengrass コアデバイスは次のデフォルトリソースを作成して使用します。

- IAM ロール: `GreengrassV2TokenExchangeRole`

  この役割には、`GreengrassV2TokenExchangeRoleAccess` という名前のポリシーと、`credentials.iot.amazonaws.com` がロールを継承することができる信頼関係があります。ポリシーには、コアデバイスに対する最低限のアクセス許可が含まれます。

> **Important**
>
> このポリシーには、S3 バケット内のファイルへのアクセスは含まれません。コアデバイスが S3 バケットからコンポーネントアーティファクトを取得できるように、ロールにアクセス許可を追加する必要があります。詳細については、「コンポーネントのアーティファクトの S3 バケットへのアクセスを許可する」を参照してください。

- AWS IoT ロールエイリアス: `GreengrassV2TokenExchangeRoleAlias`

  このロールエイリアスは IAM ロールを参照します。

詳細については、「ステップ 3: AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。

既存のコアデバイスにロールエイリアスを設定することもできます。設定するには、Greengrass nucleus コンポーネントの `iotRoleAlias` 設定パラメータを設定します。

この IAM ロールの一時的な AWS 認証情報を取得することで、カスタムコンポーネントで AWS 操作を実行することができるようになります。詳細については、「AWS サービスとやり取り」を参照してください。

トピック

- コアデバイスに対するサービスロールのアクセス許可
- コンポーネントのアーティファクトの S3 バケットへのアクセスを許可する

コアデバイスに対するサービスロールのアクセス許可

このロールにより、次のサービスがロールを継承できます。

- `credentials.iot.amazonaws.com`

AWS IoT Greengrass コアソフトウェアを使用してこのロールを作成すると、次のアクセス許可ポリシーを使用して、コアデバイスが AWS に接続し、ログを送信できるようにします。ポリシーの名前は、デフォルトで `Access` で終わる IAM ロールの名前になります。例えば、デフォルトの IAM ロール名を使用した場合には、このポリシーの名前は `GreengrassV2TokenExchangeRoleAccess` になります。

Greengrass nucleus v2.5.0 and later

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
}
```

v2.4.x

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:DescribeCertificate",
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
}
```

Earlier than v2.4.0

```
{
  "Version": "2012-10-17",
```

```
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:DescribeCertificate",
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "iot:Connect",
        "iot:Publish",
        "iot:Subscribe",
        "iot:Receive",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
}
```

コンポーネントのアーティファクトの S3 バケットへのアクセスを許可する

コアデバイスのロールは、デフォルトではコアデバイスが S3 バケットにアクセスすることを許可しません。S3 バケットにアーティファクトを持つコンポーネントをデプロイするには、コアデバイスがコンポーネントアーティファクトをダウンロードすることを許可する `s3:GetObject` アクセス許可を追加する必要があります。コアデバイスに新しいポリシーを追加することで、このアクセス許可を付与できます。

Amazon S3 のコンポーネントアーティファクトへのアクセスを許可するポリシーを追加するには

1. `component-artifact-policy.json` という名前のファイルを作成して、次の JSON をファイルにコピーします。このポリシーは、S3 バケット内のすべてのファイルへのアクセスを許可します。*`DOC-EXAMPLE-BUCKET`* を S3 バケットの名前に置き換えて、コアデバイスにアクセスを許可します。

   ```
   {
     "Version": "2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Action": [
   ```

```
        "s3:GetObject"
      ],
      "Resource": "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*"
    }
  ]
}
```

2. 次のコマンドを実行して、component-artifact-policy.json のポリシードキュメントからポリシーを作成します。

   Linux or Unix

   ```
   aws iam create-policy \
     --policy-name MyGreengrassV2ComponentArtifactPolicy \
     --policy-document file://component-artifact-policy.json
   ```

   Windows Command Prompt (CMD)

   ```
   aws iam create-policy ^
     --policy-name MyGreengrassV2ComponentArtifactPolicy ^
     --policy-document file://component-artifact-policy.json
   ```

   PowerShell

   ```
   aws iam create-policy `
     --policy-name MyGreengrassV2ComponentArtifactPolicy `
     --policy-document file://component-artifact-policy.json
   ```

   出力のポリシーメタデータから、ポリシーの Amazon リソースネーム (ARN) をコピーします。この ARN を使用して、次の手順で、このポリシーをコアデバイスのロールにアタッチします。

3. 次のコマンドを実行して、ポリシーをコアデバイスのロールにアタッチします。*GreengrassV2TokenExchangeRole* を、AWS IoT Greengrass コアソフトウェアを実行したときに指定したロールの名前に置き換えます。その後、ポリシー ARN を、前のステップで書き留めた ARN に置き換えます。

   Linux or Unix

   ```
   aws iam attach-role-policy \
     --role-name GreengrassV2TokenExchangeRole \
   ```

```
  --policy-arn
 arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
```

Windows Command Prompt (CMD)

```
aws iam attach-role-policy ^
  --role-name GreengrassV2TokenExchangeRole ^
  --policy-arn
 arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
```

PowerShell

```
aws iam attach-role-policy `
  --role-name GreengrassV2TokenExchangeRole `
  --policy-arn
 arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
```

コマンドに出力がない場合は成功したことを意味しており、コアデバイスがこの S3 バケットにアップロードしたアーティファクトにアクセスできるようになります。

インストーラがリソースをプロビジョニングするための最小限の IAM ポリシー

AWS IoT Greengrass コアソフトウェアをインストールするとき、デバイスの AWS IoT モノと IAM ロールなど、必要な AWS リソースをプロビジョニングできます。ローカル開発ツールをデバイスにデプロイすることもできます。インストーラは、AWS アカウント にこれらのアクションを実行できるように、AWS 認証情報が必要です。詳細については、「AWS IoT Greengrass Core ソフトウェアをインストールします。」を参照してください。

次のポリシーの例には、インストーラがこれらのリソースをプロビジョニングするため、必要な最小限の一連のアクションが含まれています。これらの権限は、インストーラの `--provision` 引数を指定する場合は必要です。*`account-id`* を AWS アカウント ID に置き換えて、*`GreengrassV2TokenExchangeRole`* を `--tes-role-name` インストーラ因数で指定したトークン交換ロールの名前に置き換えます。

> **Note**
>
> DeployDevTools ポリシーステートメントは、インストーラの --deploy-dev-tools 引数を指定する場合のみ必要です。

Greengrass nucleus v2.5.0 and later

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "CreateTokenExchangeRole",
            "Effect": "Allow",
            "Action": [
                "iam:AttachRolePolicy",
                "iam:CreatePolicy",
                "iam:CreateRole",
                "iam:GetPolicy",
                "iam:GetRole",
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::account-id:role/GreengrassV2TokenExchangeRole",
                "arn:aws:iam::account-id:policy/GreengrassV2TokenExchangeRoleAccess"
            ]
        },
        {
            "Sid": "CreateIoTResources",
            "Effect": "Allow",
            "Action": [
                "iot:AddThingToThingGroup",
                "iot:AttachPolicy",
                "iot:AttachThingPrincipal",
                "iot:CreateKeysAndCertificate",
                "iot:CreatePolicy",
                "iot:CreateRoleAlias",
                "iot:CreateThing",
                "iot:CreateThingGroup",
                "iot:DescribeEndpoint",
                "iot:DescribeRoleAlias",
                "iot:DescribeThingGroup",
                "iot:GetPolicy"
```

```
            ],
            "Resource": "*"
        },
        {
            "Sid": "DeployDevTools",
            "Effect": "Allow",
            "Action": [
                "greengrass:CreateDeployment",
                "iot:CancelJob",
                "iot:CreateJob",
                "iot:DeleteThingShadow",
                "iot:DescribeJob",
                "iot:DescribeThing",
                "iot:DescribeThingGroup",
                "iot:GetThingShadow",
                "iot:UpdateJob",
                "iot:UpdateThingShadow"
            ],
            "Resource": "*"
        }
    ]
}
```

Earlier than v2.5.0

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "CreateTokenExchangeRole",
            "Effect": "Allow",
            "Action": [
                "iam:AttachRolePolicy",
                "iam:CreatePolicy",
                "iam:CreateRole",
                "iam:GetPolicy",
                "iam:GetRole",
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::account-id:role/GreengrassV2TokenExchangeRole",
                "arn:aws:iam::account-
id:policy/GreengrassV2TokenExchangeRoleAccess",
```

```
                "arn:aws:iam::aws:policy/GreengrassV2TokenExchangeRoleAccess"
            ]
        },
        {
            "Sid": "CreateIoTResources",
            "Effect": "Allow",
            "Action": [
                "iot:AddThingToThingGroup",
                "iot:AttachPolicy",
                "iot:AttachThingPrincipal",
                "iot:CreateKeysAndCertificate",
                "iot:CreatePolicy",
                "iot:CreateRoleAlias",
                "iot:CreateThing",
                "iot:CreateThingGroup",
                "iot:DescribeEndpoint",
                "iot:DescribeRoleAlias",
                "iot:DescribeThingGroup",
                "iot:GetPolicy"
            ],
            "Resource": "*"
        },
        {
            "Sid": "DeployDevTools",
            "Effect": "Allow",
            "Action": [
                "greengrass:CreateDeployment",
                "iot:CancelJob",
                "iot:CreateJob",
                "iot:DeleteThingShadow",
                "iot:DescribeJob",
                "iot:DescribeThing",
                "iot:DescribeThingGroup",
                "iot:GetThingShadow",
                "iot:UpdateJob",
                "iot:UpdateThingShadow"
            ],
            "Resource": "*"
        }
    ]
}
```

Greengrass サービスロール

Greengrass サービスロールは、ユーザーに代わって AWS サービスからリソースへのアクセスを AWS IoT Greengrass に許可する AWS Identity and Access Management (IAM) サービスロールです。このロールにより、AWS IoT Greengrass でクライアントデバイスの ID を確認し、コアデバイスの接続情報を管理できるようになります。

> **Note**
>
> また、AWS IoT Greengrass V1 はこのロールを使用して重要なタスクを実行します。詳細については、「AWS IoT Greengrass V1 デベロッパーガイド」の「Greengrass サービスロール」を参照してください。

AWS IoT Greengrass がリソースにアクセスすることを許可するには、AWS アカウント にGreengrass サービスロールが関連付けられていて、信頼されたエンティティとして AWS IoT Greengrass が指定されている必要があります。ロールには、AWSGreengrassResourceAccessRolePolicyマネージドポリシー、または使用するAWS IoT Greengrass機能に対する同等のアクセス許可を定義するカスタムポリシーを含める必要があります。 はこのポリシーAWSを維持し、 が AWSリソースにアクセスAWS IoT Greengrassするために使用するアクセス許可のセットを定義します。詳細については、「AWS マネージドポリシー: AWSGreengrassResourceAccessRolePolicy」を参照してください。

AWS リージョン で同じ Greengrass サービスロールを再利用できますが、AWS IoT Greengrass を使用するすべての AWS リージョン でユーザーのアカウントに関連付けられている必要があります。サービスロールが現在の AWS リージョン で設定されていない場合、コアデバイスはクライアントデバイスの検証に失敗し、接続情報の更新に失敗します。

以下のセクションでは、AWS Management Console または AWS CLI で Greengrass サービスロールを作成して管理する方法について説明します。

トピック

- Greengrass サービスロールを管理する (コンソール)
- Greengrass サービスロールを管理する (CLI)
- 以下も参照してください。

> **Note**
>
> サービスレベルのアクセスを承認するサービスロールに加えて、Greengrass コアデバイスにトークン交換サービスを割り当てます。トークン交換ロールは、コアデバイス上の Greengrass コンポーネントと Lambda 関数が AWS のサービスにどのようにアクセスできるかを制御する別の IAM ロールです。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

Greengrass サービスロールを管理する (コンソール)

AWS IoT コンソールを使用すると、Greengrass サービスロールを簡単に管理できます。たとえば、コアデバイスに対してクライアントデバイスの検出を設定すると、コンソールでユーザーの AWS アカウント が現在の AWS リージョン の Greengrass サービスロールに添付されているかどうかの確認が行われます。アタッチされていない場合、コンソールによるサービスロールの作成および設定が可能です。詳細については、「the section called “Greengrass サービスロールを作成する”」を参照してください。

コンソールは以下のロール管理タスクに使用できます。

トピック

- Greengrass サービスロールを見つける (コンソール)
- Greengrass サービスロールを作成する (コンソール)
- Greengrass サービスロールを変更する (コンソール)
- Greengrass サービスロールをデタッチする (コンソール)

> **Note**
>
> コンソールにサインインするユーザーには、サービスロールを表示、作成、または変更するためのアクセス許可が必要です。

Greengrass サービスロールを見つける (コンソール)

以下の手順を使用して、現在の AWS リージョン で AWS IoT Greengrass が使用しているサービスロールを探します。

1. AWS IoT コンソールに移動します。
2. ナビゲーションペインで [設定] を選択します。
3. [Greengrass service role (Greengrass サービスロール)] セクションまでスクロールして、サービスロールとそのポリシーを表示します。

   サービスロールが表示されない場合は、コンソールでサービスロールの作成または設定が可能です。詳細については、「Greengrass サービスロールを作成する」を参照してください。

Greengrass サービスロールを作成する (コンソール)

コンソールによるデフォルトの Greengrass サービスロールの作成と設定が可能です。このロールには以下のプロパティがあります。

プロパティ	値
名前	`Greengrass_ServiceRole`
信頼されたエンティティ	`AWS service: greengrass`
ポリシー	AWSGreengrassResourceAccessRolePolicy

> **Note**
>
> このロールをAWS IoT Greengrass V1 デバイスセットアップスクリプトで作成する場合、ロール名は `GreengrassServiceRole_random-string` になります。

コアデバイスのクライアントデバイス検出を設定すると、現在の AWS リージョン でユーザーの AWS アカウント に Greengrass サービスロール が関連付けられているかどうかを、コンソールが確認します。関連付けられていない場合、コンソールでは、AWS のサービスに対してユーザーに代わって読み書きすることを AWS IoT Greengrass に許可するように求められます。

許可を付与すると、コンソールでは、AWS アカウント に `Greengrass_ServiceRole` という名前のロールがあるかどうかの確認が行われます。

- そのロールがある場合、コンソールで、そのサービスロールが現在の AWS リージョン の AWS アカウント にアタッチされます。

- そのロールがない場合、コンソールで、デフォルトの Greengrass サービスロールが作成され、現在の AWS リージョン の AWS アカウント にアタッチされます。

> **Note**
>
> カスタムロールポリシーを使用してサービスロールを作成する場合は、IAM コンソールを使用してロールを作成または変更します。詳細については、「IAM ユーザーガイド」の「AWS サービスにアクセス許可を委任するロールの作成」または「ロールの修正」を参照してください。使用する機能およびリソースの `AWSGreengrassResourceAccessRolePolicy` マネージドポリシーと同等のアクセス許可が、ロールによって付与されることを確認します。また、信頼ポリシーには、`aws:SourceArn` および `aws:SourceAccount` グローバル条件コンテキストキーも含めて、混乱した代理によるセキュリティ問題を防止することをお勧めします。条件コンテキストキーを使用すると、指定したアカウントと Greengrass ワークスペースからのリクエストのみを許可するようにアクセスを制限できます。混乱した代理に関する問題の詳細については、「サービス間の混乱した代理の防止」を参照してください。
> サービスロールを作成する場合は、AWS IoT コンソールに戻り、ロールを AWS アカウント にアタッチします。これは、[Settings] (設定) ページの [Greengrass service role] (Greengrass サービスロール) で行えます。

Greengrass サービスロールを変更する (コンソール)

以下の手順を使用して、コンソールで現在選択されている AWS リージョン の AWS アカウント にアタッチする別の Greengrass サービスロールを選択します。

1. AWS IoT コンソールに移動します。
2. ナビゲーションペインで [設定] を選択します。
3. [Greengrass service role] (Greengrass サービスロール) で、[Choose different role] (別のロールの選択) を選択します。

   [Update Greengrass service role] (Greengrass サービスロールを更新) ダイアログボックスが開き、AWS IoT Greengrass を信頼するエンティティとして定義する AWS アカウント の IAM ロールが表示されます。
4. アタッチする Greengrass サービスロールを選択します。
5. [Attach role] (ロールをアタッチする)を選択します。

Greengrass サービスロールをデタッチする (コンソール)

コンソールで現在選択されている AWS リージョン の AWS アカウントから Greengrass サービスロールをデタッチするには、以下の手順を使用します。これにより、現在の AWS リージョン の AWS のサービスに対する AWS IoT Greengrass の許可が取り消されます。

> **Important**
>
> サービスロールをデタッチすると、アクティブなオペレーションが中断される場合があります。

1. AWS IoT コンソールに移動します。
2. ナビゲーションペインで [設定] を選択します。
3. [Greengrass service role] (Greengrass サービスロール) で、[Detach role] (ロールのデタッチ) を選択します。
4. 確認ダイアログボックスで、[Detach] (デタッチ) を選択します。

> **Note**
>
> ロールが不要になった場合は、IAM コンソールで削除できます。詳細については、「IAM ユーザーガイド」の「ロールまたはインスタンスプロファイルを削除する」を参照してください。
> 他のロールで、AWS IoT Greengrass にお客様のリソースへのアクセスを許可している可能性があります。ユーザーに代わってアクセス権限を引き受けることを AWS IoT Greengrass に許可するロールをすべて見つけるには、IAM コンソールの [Roles] (ロール) ページにある [Trusted entities] (信頼済みエンティティ) 列で、[AWS service: greengrass] (サービス: greengrass) を含むロールを探します。

Greengrass サービスロールを管理する (CLI)

次の手順では、AWS Command Line Interface がインストールされていて AWS アカウント を使用するように設定されていることを前提としています。詳細については、「AWS Command Line Interface ユーザーガイド」の「AWS CLI のインストール、更新、アンインストール」と「AWS CLI の設定」を参照してください。

AWS CLI は以下のロール管理タスクに使用できます。

トピック

- Greengrass サービスロールを取得する (CLI)
- Greengrass サービスロールを作成する (CLI)
- Greengrass サービスロールを削除する (CLI)

Greengrass サービスロールを取得する (CLI)

Greengrass サービスロールが AWS リージョン 内の AWS アカウント に関連付けられているかどうか調べるには、以下の手順を使用します。

- サービスロールを取得します。*region* を AWS リージョン に置き換えます (例:us-west-2)。

  ```
  aws greengrassv2 get-service-role-for-account --region region
  ```

  Greengrass サービスロールが既にアカウントに関連付けられている場合、リクエストに対して以下のロールメタデータが返されます。

  ```
  {
    "associatedAt": "timestamp",
    "roleArn": "arn:aws:iam::account-id:role/path/role-name"
  }
  ```

  リクエストしてもロールメタデータが返されない場合は、サービスロールを作成し (存在しない場合)、AWS リージョン 内でアカウントに関連付ける必要があります。

Greengrass サービスロールを作成する (CLI)

次のステップを使用してロールを作成し、AWS アカウント に関連付けます。

IAM を使用して、サービスロールを作成するには

1. AWS IoT Greengrass がロールを継承できるように許可する信頼ポリシーを持つロールを作成します。この例では、Greengrass_ServiceRole という名前のロールを作成しますが、別の名前を使用できます。また、信頼ポリシーには、aws:SourceArn および aws:SourceAccount グローバル条件コンテキストキーも含めて、混乱した代理によるセキュリティ問題を防止することをお勧めします。条件コンテキストキーを使用すると、指定したアカウントと Greengrass ワークスペースからのリクエストのみを許可するようにアクセスを制限できます。混乱した代理に関する問題の詳細については、「サービス間の混乱した代理の防止」を参照してください。

Linux or Unix

```
aws iam create-role --role-name Greengrass_ServiceRole --assume-role-policy-
document '{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "greengrass.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnLike": {
          "aws:SourceArn": "arn:aws:greengrass:region:account-id:*"
        },
        "StringEquals": {
          "aws:SourceAccount": "account-id"
        }
      }
    }
  ]
}'
```

Windows Command Prompt (CMD)

```
aws iam create-role --role-name Greengrass_ServiceRole --assume-role-
policy-document "{\"Version\":\"2012-10-17\",\"Statement\":[{\"Effect
\":\"Allow\",\"Principal\":{\"Service\":\"greengrass.amazonaws.com\"},
\"Action\":\"sts:AssumeRole\",\"Condition\":{\"ArnLike\":{\"aws:SourceArn
\":\"arn:aws:greengrass:region:account-id:*\"},\"StringEquals\":
{\"aws:SourceAccount\":\"account-id\"}}}]}"
```

PowerShell

```
aws iam create-role --role-name Greengrass_ServiceRole --assume-role-policy-
document '{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
```

```
      "Principal": {
        "Service": "greengrass.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnLike": {
          "aws:SourceArn": "arn:aws:greengrass:region:account-id:*"
        },
        "StringEquals": {
          "aws:SourceAccount": "account-id"
        }
      }
    }
  ]
}'
```

2. 出力のロールメタデータからロールの ARN をコピーします。ARN を使用して、ロールをアカウントに関連付けます。
3. AWSGreengrassResourceAccessRolePolicy ポリシーをロールにアタッチします。

```
aws iam attach-role-policy --role-name Greengrass_ServiceRole --policy-arn
 arn:aws:iam::aws:policy/service-role/AWSGreengrassResourceAccessRolePolicy
```

AWS アカウント にサービスロールを関連付けるには

- ロールとアカウントを関連付けます。*role-arn* をサービスロール ARN と置き換え、*region* を AWS リージョン (us-west-2 など) と置き換えます。

```
aws greengrassv2 associate-service-role-to-account --role-arn role-arn --
region region
```

成功すると、リクエストは、以下のようなレスポンスを返します。

```
{
  "associatedAt": "timestamp"
}
```

Greengrass サービスロールを削除する (CLI)

次のステップを使用して、Greengrass サービスロールの関連付けを AWS アカウント から解除します。

- アカウントからサービスロールの関連付けを解除します。*region* を AWS リージョン に置き換えます (例:us-west-2)。

```
aws greengrassv2 disassociate-service-role-from-account --region region
```

成功すると、以下のレスポンスが返されます。

```
{
  "disassociatedAt": "timestamp"
}
```

> **Note**
>
> 任意の AWS リージョン で使用していない場合は、サービスロールを削除する必要があります。最初に、delete-role-policy を使用して AWSGreengrassResourceAccessRolePolicy 管理ポリシーをロールからデタッチし、次に delete-role を使用してロールを削除します。詳細については、「IAM ユーザーガイド」の「ロールまたはインスタンスプロファイルを削除する」を参照してください。

以下も参照してください。

- 詳細については、「IAM ユーザーガイド」の「AWS のサービスにアクセス許可を委任するロールの作成」を参照してください。
- 「IAM ユーザーガイド」の「ロールの修正」
- 「IAM ユーザーガイド」の「ロールまたはインスタンスプロファイルを削除する」
- AWS CLI コマンドリファレンスで使用可能な AWS IoT Greengrass コマンド
  - associate-service-role-toアカウント
  - disassociate-service-role-from- アカウント
  - get-service-role-for- アカウント

- AWS CLI コマンドリファレンスで使用可能な IAM コマンド
  - attach-role-policy
  - create-role
  - delete-role
  - delete-role-policy

AWS の AWS IoT Greengrass 管理ポリシー

AWS マネージドポリシーは、AWS が作成および管理するスタンドアロンポリシーです。AWS マネージドポリシーは、多くの一般的なユースケースでアクセス許可を提供するように設計されているため、ユーザー、グループ、ロールへのアクセス許可の割り当てを開始できます。

AWS マネージドポリシーは、特定のユースケースに対して最小特権のアクセス許可を付与しない場合があることに注意してください。AWS のすべてのお客様が使用できるようになることを避けるためです。ユースケース別にカスタマーマネージドポリシーを定義することで、アクセス許可を絞り込むことをお勧めします。

AWS 管理ポリシーで定義したアクセス権限は変更できません。AWS が AWS マネージドポリシーに定義されているアクセス許可を更新すると、更新はポリシーがアタッチされているすべてのプリンシパルアイデンティティ (ユーザー、グループ、ロール) に影響します。新しい AWS のサービスを起動するか、既存のサービスで新しい API 操作が使用可能になると、AWS が AWS マネージドポリシーを更新する可能性が最も高くなります。

詳細については、「IAM ユーザーガイド」の「AWS 管理ポリシー」を参照してください。

トピック

- AWS マネージドポリシー: AWSGreengrassFullAccess
- AWS マネージドポリシー:　AWSGreengrassReadOnlyAccess
- AWS マネージドポリシー:　AWSGreengrassResourceAccessRolePolicy
- AWS IoT Greengrass マネージドポリシーの AWS 更新

AWS マネージドポリシー: AWSGreengrassFullAccess

`AWSGreengrassFullAccess` ポリシーは IAM ID にアタッチできます。

このポリシーは、プリンシパルにすべての AWS IoT Greengrass アクションへのフルアクセスを許可する管理者権限を付与します。

許可の詳細

このポリシーには、以下の許可が含まれています。

- greengrass – プリンシパルに AWS IoT Greengrass のすべてのアクションへのフルアクセスを許可します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:*"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS マネージドポリシー: AWSGreengrassReadOnlyAccess

AWSGreengrassReadOnlyAccess ポリシーは IAM ID にアタッチできます。

このポリシーにより、プリンシパルは AWS IoT Greengrass の情報を表示できるが、変更できないようにする読み取り専用のアクセス許可が付与されます。例えば、これらのアクセス許可を持つプリンシパルは、Greengrass コアデバイスにデプロイされたコンポーネントのリストを表示できますが、そのデバイスで実行されるコンポーネントを変更するためのデプロイを作成することはできません。

許可の詳細

このポリシーには、以下の許可が含まれています。

- greengrass - プリンシパルは、アイテムのリストまたはアイテムに関する詳細を返すアクションを実行することができます。これには、List または Get で始まる API オペレーションが含まれます。

```
{
    "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:List*",
                "greengrass:Get*"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS マネージドポリシー: AWSGreengrassResourceAccessRolePolicy

`AWSGreengrassResourceAccessRolePolicy` ポリシーを IAM エンティティにアタッチできます。AWS IoT Greengrass はまた、AWS IoT Greengrass がユーザーに代わりアクションを実行することを可能にするサービスロールをこのポリシーにアタッチします。詳細については、「Greengrass サービスロール」を参照してください。

このポリシーにより、Lambda 関数の取得、AWS IoT デバイスシャドウの管理、Greengrass クライアントデバイスの検証など、AWS IoT Greengrass に重要なタスクの実行を許可する管理者権限が付与されます。

許可の詳細

このポリシーには、以下の許可が含まれています。

- `greengrass` - Greengrass リソースを管理します。
- `iot` (*Shadow) - 名前に次の特別な識別子が含まれる AWS IoT シャドウを管理します。これらのアクセス許可は、AWS IoT Greengrass がコアデバイスと通信できるようにするために必要です。
  - `*-gci` – AWS IoT Greengrass は、このシャドウを使用してコアデバイスの接続情報を保存し、クライアントデバイスがコアデバイスを検出して接続できるようにします。
  - `*-gcm` – AWS IoT Greengrass V1 は、このシャドウを使用して、Greengrass グループの認証局 (CA) 証明書がローテーションされたことをコアデバイスに通知します。
  - `*-gda` – AWS IoT Greengrass V1 は、このシャドウを使用して、コアデバイスにデプロイを通知します。
  - `GG_*` – 未使用。
- `iot` (`DescribeThing` および `DescribeCertificate`) - AWS IoT モノおよび証明書に関する情報を取得します。これらのアクセス許可は、AWS IoT Greengrass がコアデバイスに接続するクラ

イアントデバイスを検証できるようにするために必要です。詳細については、「ローカル IoT デバイスとやり取りする」を参照してください。

- `lambda` - AWS Lambda 関数に関する情報を取得します。このアクセス許可は、AWS IoT Greengrass V1 がLambda 関数を Greengrass コアにデプロイできるようにするために必要です。詳細については、「AWS IoT Greengrass V1 デベロッパーガイド」の「AWS IoT Greengrass コアの Lambda 関数の実行」を参照してください。
- `secretsmanager` - 名前が `greengrass-` で始まる AWS Secrets Manager シークレットの値を取得します。このアクセス許可は、AWS IoT Greengrass V1 が Secrets Manager シークレットを Greengrass コアにデプロイできるようにするために必要です。詳細については、[AWS IoT Greengrass V1 Developer Guide] (デベロッパーガイド) の[Deploy secrets to the AWS IoT Greengrass core] (Core にシークレットをデプロイする) を参照してください。
- `s3` - 名前に `greengrass` または `sagemaker` が含まれる S3 バケットからファイルオブジェクトを取得します。これらのアクセス許可は、S3 バケットに保存する機械学習リソースを AWS IoT Greengrass V1 がデプロイできるようにするために必要です。詳細については、「AWS IoT Greengrass V1 デベロッパーガイド」の「機械学習リソース」を参照してください。
- `sagemaker` - Amazon SageMaker 機械学習推論モデルに関する情報を取得します。このアクセス許可は、AWS IoT Greengrass V1 が ML モデルを Greengrass コアにデプロイできるようにするために必要です。詳細については、「AWS IoT Greengrass V1 デベロッパーガイド」の「[Perform machine learning inference] (機械学習の推論を実行する)」を参照してください。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowGreengrassAccessToShadows",
            "Action": [
                "iot:DeleteThingShadow",
                "iot:GetThingShadow",
                "iot:UpdateThingShadow"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:iot:*:*:thing/GG_*",
                "arn:aws:iot:*:*:thing/*-gcm",
                "arn:aws:iot:*:*:thing/*-gda",
                "arn:aws:iot:*:*:thing/*-gci"
            ]
        },
```

```
        {
            "Sid": "AllowGreengrassToDescribeThings",
            "Action": [
                "iot:DescribeThing"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:iot:*:*:thing/*"
        },
        {
            "Sid": "AllowGreengrassToDescribeCertificates",
            "Action": [
                "iot:DescribeCertificate"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:iot:*:*:cert/*"
        },
        {
            "Sid": "AllowGreengrassToCallGreengrassServices",
            "Action": [
                "greengrass:*"
            ],
            "Effect": "Allow",
            "Resource": "*"
        },
        {
            "Sid": "AllowGreengrassToGetLambdaFunctions",
            "Action": [
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration"
            ],
            "Effect": "Allow",
            "Resource": "*"
        },
        {
            "Sid": "AllowGreengrassToGetGreengrassSecrets",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:secretsmanager:*:*:secret:greengrass-*"
        },
        {
            "Sid": "AllowGreengrassAccessToS3Objects",
            "Action": [
```

```
                "s3:GetObject"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:s3:::*Greengrass*",
                "arn:aws:s3:::*GreenGrass*",
                "arn:aws:s3:::*greengrass*",
                "arn:aws:s3:::*Sagemaker*",
                "arn:aws:s3:::*SageMaker*",
                "arn:aws:s3:::*sagemaker*"
            ]
        },
        {
            "Sid": "AllowGreengrassAccessToS3BucketLocation",
            "Action": [
                "s3:GetBucketLocation"
            ],
            "Effect": "Allow",
            "Resource": "*"
        },
        {
            "Sid": "AllowGreengrassAccessToSageMakerTrainingJobs",
            "Action": [
                "sagemaker:DescribeTrainingJob"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:sagemaker:*:*:training-job/*"
            ]
        }
    ]
}
```

AWS IoT Greengrass マネージドポリシーの AWS 更新

このサービスがこれらの変更の追跡を開始した時点から、AWS IoT Greengrass の AWS のマネージドポリシーの更新に関する詳細を表示できます。このページの変更に関する自動通知については、AWS IoT Greengrass V2 ドキュメントの履歴ページの RSS フィードをサブスクライブしてください。

変更	説明	日付
AWS IoT Greengrass は変更の追跡を開始しました	AWS IoT Greengrass が AWS マネージドポリシーの変更の追跡を開始しました。	2021 年 7 月 2 日

サービス間の混乱した代理の防止

混乱した代理問題は、アクションを実行する許可を持たないエンティティが、より特権のあるエンティティにアクションを実行するように強制できるセキュリティの問題です。AWS では、サービス間でのなりすましが、混乱した代理問題を生じさせることがあります。サービス間でのなりすましは、1 つのサービス (呼び出し元サービス) が、別のサービス (呼び出し対象サービス) を呼び出すときに発生する可能性があります。呼び出し元サービスは、本来ならアクセスすることが許可されるべきではない方法でその許可を使用して、別の顧客のリソースに対する処理を実行するように操作される場合があります。これを防ぐため、AWS では、アカウント内のリソースへのアクセス権が付与されたサービスプリンシパルですべてのサービスのデータを保護するために役立つツールを提供しています。

リソースポリシーで `aws:SourceArn` および `aws:SourceAccount` のグローバル条件コンテキストキーを使用して、AWS IoT Greengrass が別のサービスに付与する許可をそのリソースに制限することをお勧めします。両方のグローバル条件コンテキストキーを使用しており、それらが同じポリシーステートメントで使用されるときは、`aws:SourceAccount` 値と、`aws:SourceArn` 値のアカウントが同じアカウント ID を使用する必要があります。

`aws:SourceArn` の値は、`sts:AssumeRole` リクエストに関連付けられている Greengrass のカスタマーリソースである必要があります。

混乱した代理問題から保護するための最も効果的な方法は、リソースの完全な ARN を指定しながら、`aws:SourceArn` グローバル条件コンテキストキーを使用することです。リソースの完全な ARN が不明な場合や、複数のリソースを指定する場合には、グローバルコンテキスト条件キー `aws:SourceArn` で、ARN の未知部分を示すためにワイルドカード (*) を使用します。例: `arn:aws:greengrass::account-id:*`。

`aws:SourceArn` と `aws:SourceAccount` グローバル条件コンテキストキーを使用するポリシーの例については、「Greengrass サービスロールを作成する」を参照してください。

AWS IoT Greengrass のアイデンティティとアクセスの問題のトラブルシューティング

次の情報は、AWS IoT Greengrass と IAM の使用に伴って発生する可能性がある一般的な問題の診断や修復に役立ちます。

問題点

- AWS IoT Greengrass でアクションを実行する権限がありません。
- iam:PassRole を実行する権限がない
- 管理者として AWS IoT Greengrass へのアクセスを他のユーザーに許可したい
- 自分の AWS アカウント 以外のユーザーに AWS IoT Greengrass リソースへのアクセスを許可したい

一般的なトラブルシューティングヘルプについては、「トラブルシューティング」を参照してください。

AWS IoT Greengrass でアクションを実行する権限がありません。

アクションを実行する権限がないというエラーが表示された場合、管理者に問い合わせてサポートを依頼する必要があります。お客様のユーザー名とパスワードを発行したのが、担当の管理者です。

以下の例のエラーは、`mateojackson` IAM ユーザーがコアデバイスの詳細を表示しようとしているが、`greengrass:GetCoreDevice` アクセス許可がない場合に発生します。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to
 perform: greengrass:GetCoreDevice on resource: arn:aws:greengrass:us-
west-2:123456789012:coreDevices/MyGreengrassCore
```

この場合、Mateo は、`greengrass:GetCoreDevice` アクションを使用して `arn:aws:greengrass:us-west-2:123456789012:coreDevices/MyGreengrassCore` リソースへのアクセスが許可されるように、管理者にポリシーの更新を依頼します。

以下は、AWS IoT Greengrass を操作するときに発生する可能性がある一般的な IAM の問題です。

iam:PassRole を実行する権限がない

`iam:PassRole` アクションを実行する権限がないというエラーが表示された場合は、ポリシーを更新して AWS IoT Greengrass にロールを渡すことができるようにする必要があります。

一部の AWS のサービスでは、新しいサービスロールやサービスにリンクされたロールを作成せずに、既存のロールをサービスに渡すことができます。そのためには、サービスにロールを渡す許可が必要です。

以下の例のエラーは、 `marymajor` という IAM ユーザーがコンソールを使用して AWS IoT Greengrass でアクションを実行しようする場合に発生します。ただし、このアクションをサービスが実行するには、サービスロールから付与されたアクセス許可が必要です。Mary には、ロールをサービスに渡す許可がありません。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

この場合、メアリーのポリシーを更新してメアリーに `iam:PassRole` アクションの実行を許可する必要があります。

サポートが必要な場合は、AWS 管理者に問い合わせてください。サインイン資格情報を提供した担当者が管理者です。

管理者として AWS IoT Greengrass へのアクセスを他のユーザーに許可したい

AWS IoT Greengrass へのアクセスを他のユーザーに許可するには、アクセスを必要とする人またはアプリケーションの IAM エンティティ (ユーザーまたはロール) を作成する必要があります。ユーザーは、このエンティティの認証情報を使用して AWS にアクセスします。次に、AWS IoT Greengrass の適切なアクセス許可を付与するポリシーを、そのエンティティにアタッチする必要があります。

すぐに開始するには、IAM ユーザーガイドの「IAM が委任した最初のユーザーおよびグループの作成」を参照してください。

自分の AWS アカウント 以外のユーザーに AWS IoT Greengrass リソースへのアクセスを許可したい

他のアカウントのユーザーや組織外のユーザーが、AWS リソースへのアクセスに使用できる IAM ロールを作成できます。ロールを引き受けるように信頼されたユーザーを指定することができます。詳細については、「IAM ユーザーガイド」の「Providing access to an IAM user in another AWS アカウント that you own」(所有している別の AWS アカウント へのアクセス権を IAM ユーザーに提供) と「Providing access to AWS アカウントs owned by third parties」(第三者が所有する AWS アカウント へのアクセス権を提供) を参照してください。

AWS IoT Greengrass は、リソースベースのポリシーまたはアクセスコントロールリスト (ACL) に基づくクロスアカウントアクセスをサポートしていません。

プロキシまたはファイアウォールを介したデバイストラフィックを許可する

Greengrass コアデバイスと Greengrass コンポーネントは、AWS サービスや他のウェブサイトに対するアウトバウンドリクエストを実行します。セキュリティ対策として、アウトバウンドトラフィックを、狭い範囲のエンドポイントとポートに制限することがあります。エンドポイントとポートに関する以下の情報を活用することで、プロキシ、ファイアウォール、または Amazon VPC セキュリティグループを通じてデバイストラフィックを制限することができます。プロキシを使用するようにコアデバイスを設定する方法の詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

トピック

- 基本操作のためのエンドポイント
- 自動プロビジョニングを使用したインストール向けのエンドポイント
- AWS から提供されたコンポーネント向けのエンドポイント

基本操作のためのエンドポイント

Greengrass コアデバイスは、基本的な操作に次のエンドポイントとポートを使用します。

AWS IoT エンドポイントの取得

AWS アカウント に AWS IoT エンドポイントを入手して、後で使用するために保存してください。デバイスはこれらのエンドポイントを使用して AWS IoT に接続します。以下の操作を実行します。

1. AWS アカウント の AWS IoT データエンドポイントを取得します。

   ```
   aws iot describe-endpoint --endpoint-type iot:Data-ATS
   ```

   要求が正常に処理された場合、レスポンスは次の例のようになります。

   ```
   {
     "endpointAddress": "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
   ```

```
}
```

2. AWS IoT の AWS アカウント 認証情報エンドポイントを取得します。

```
aws iot describe-endpoint --endpoint-type iot:CredentialProvider
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "endpointAddress": "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
}
```

エンドポイント	ポート	必要	説明
greengrass-ats.iot. *region*.amazonaws.com	8443 または 443	はい	デプロイのインストールやクライアントデバイスの操作などの、データプレーンオペレーションに使用されます。
device-data-prefix -ats.iot. *region*.amazonaws.com	MQTT: 8883 または 443 HTTPS: 8443 または 443	はい	MQTT 通信や AWS IoT Core とのシャドウ同期など、デバイスを管理するためのデータプレーンオペ

エンドポイント	ポート	必要	説明
			レーションに使用されます。
device-credentials-prefix .credentials.iot. *region*.amazonaws.com	443	はい	コアデバイスが Amazon S3 からコンポーネントアーティファクトをダウンロードしたり、その他の操作を行うために使用する AWS 認証情報を取得するために使用します。詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

エンドポイント	ポート	必要	説明
*.s3.amazonaws.com *.s3.*region*.amazonaws.com	443	はい	デプロイに使用されます。エンドポイントのプレフィックスは内部的に制御されており、いつでも変更される可能性があるため、この形式には * 文字が含まれます。

エンドポイント	ポート	必要	説明
data.iot. *region*.amazonaws.com	443	いいえ	コアデバイスが Greengrass nucleus の v2.4.0 以前のバージョンを実行している場合に必要で、ネットワークプロキシを使用するように設定されています。コアデバイスは、プロキシの背後にあるときに、AWS IoT Core との MQTT 通信にこのエンドポイントを使用します。詳細については、「ネットワークプロキシを設定する」を参照してください。

自動プロビジョニングを使用したインストール向けのエンドポイント

Greengrass コアデバイスは、[自動リソースプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストール]するときに、次のエンドポイントとポートを使用します。

エンドポイント	ポート	必要	説明
`iot.region.amazonaws.com`	443	はい	AWS IoT リソースを作成し、既存の AWS IoT リソースに関する情報を取得するために使用します。
`iam.amazonaws.com`	443	はい	IAM リソースを作成し、既存の IAM リソースに関する情報を取得するために使用します。
`sts.region.amazonaws.com`	443	はい	AWS アカウント の ID を取得するために使用します。
`greengrass. region.amazonaws.com`	443	いいえ	`--deploy-dev-`

エンドポイント	ポート	必要	説明
			`tools` 引数を使用してGreengrass CLI コンポーネントをコアデバイスにデプロイする場合に必要です。

AWS から提供されたコンポーネント向けのエンドポイント

Greengrass コアデバイスは、実行するソフトウェアコンポーネントに応じて、追加のエンドポイントを使用します。AWS から提供された各コンポーネントが必要とするエンドポイントは、このデベロッパーガイドの各コンポーネントのページにある [Requirements] (要件) のセクションで確認することができます。詳細については、「AWS が提供したコンポーネント」を参照してください。

AWS IoT Greengrass のコンプライアンス検証

AWS のサービス が特定のコンプライアンスプログラムの対象範囲内にあるかどうかを確認するには、「コンプライアンスプログラムによる対象範囲内の AWS のサービス」を参照し、目的のコンプライアンスプログラムを選択してください。一般的な情報については、「AWS コンプライアンスプログラム」を参照してください。

AWS Artifact を使用して、サードパーティーの監査レポートをダウンロードできます。詳細については、「Downloading Reports in AWS Artifact」を参照してください。

AWS のサービスを使用する際のユーザーのコンプライアンス責任は、ユーザーのデータの機密性や貴社のコンプライアンス目的、適用される法律および規制によって決まります。AWS では、コンプライアンスに役立つ次のリソースを提供しています。

- セキュリティとコンプライアンスのクイックスタートガイド - これらのデプロイガイドでは、アーキテクチャ上の考慮事項について説明し、セキュリティとコンプライアンスに重点を置いたベースライン環境を AWS にデプロイするための手順を示します。

- 「Amazon Web Services での HIPAA のセキュリティとコンプライアンスのためのアーキテクチャ」 - このホワイトペーパーは、企業が AWS を使用して HIPAA 対象アプリケーションを作成する方法を説明しています。

> **Note**
>
> すべての AWS のサービスが HIPAA 適格であるわけではありません。詳細については、「HIPAA 対応サービスのリファレンス」を参照してください。

- AWS コンプライアンスのリソース - このワークブックおよびガイドのコレクションは、顧客の業界と拠点に適用されるものである場合があります。
- AWS Customer Compliance Guide — コンプライアンスの観点から見た責任共有モデルを理解できます。このガイドは、AWS のサービスを保護するためのベストプラクティスを要約したものであり、複数のフレームワーク (米国標準技術研究所 (NIST)、ペイメントカード業界セキュリティ標準評議会 (PCI)、国際標準化機構 (ISO) など) にわたるセキュリティ統制へのガイダンスがまとめられています。
- 「AWS Config デベロッパーガイド」の「ルールでのリソースの評価」 - AWS Config サービスは、自社のプラクティス、業界ガイドライン、および規制に対するリソースの設定の準拠状態を評価します。
- AWS Security Hub - この AWS のサービスは、AWS 内のセキュリティ状態の包括的なビューを提供します。Security Hub では、セキュリティコントロールを使用して AWS リソースを評価し、セキュリティ業界標準とベストプラクティスに対するコンプライアンスをチェックします。サポートされているサービスとコントロールのリストについては、「Security Hub のコントロールリファレンス」を参照してください。
- AWS Audit Manager - この AWS のサービスは、AWS の使用状況を継続的に監査して、リスクの管理方法や、規制および業界標準へのコンプライアンスの管理方法を簡素化するために役立ちます。

AWS IoT Greengrass での耐障害性

AWS のグローバルインフラストラクチャはアマゾン ウェブ サービスリージョンとアベイラビリティーゾーンを中心として構築されます。各 AWS リージョン には、低レイテンシー、高いスループット、そして高度の冗長ネットワークで接続されている複数の物理的に独立し隔離されたアベイラビリティーゾーンがあります。アベイラビリティーゾーンを使用すると、中断することなくゾーン間で自動的にフェイルオーバーするアプリケーションとデータベースを設計および運用できます。アベ

イラビリティーゾーンは、従来の単一または複数のデータセンターインフラストラクチャよりも可用性が高く、フォールトトレラントで、スケーラブルです。

詳細については、「AWS グローバルインフラストラクチャ」を参照してください。

AWS では、AWS IoT Greengrass グローバルインフラストラクチャに加えて、データの耐障害性とバックアップのニーズに対応できるように複数の機能を提供しています。

- ログをローカルファイルシステムと CloudWatch Logs に書き込むように、Greengrass グループを設定できます。コアデバイスが切断された場合でも、ファイルシステム上でメッセージを記録し続けることができます。再接続されると、ログメッセージが CloudWatch Logs に書き込まれます。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。
- デプロイ中にコアデバイスの電源が失われると、AWS IoT Greengrass コアソフトウェアが再び再起動した後にデプロイが再開されます。
- コアデバイスがインターネットから切断された場合でも、Greengrass クライアントデバイスはローカルネットワークを介して通信を続けることができます。
- ストリームマネージャーのストリームを読み込み、ローカルストレージの送信先にデータを送る Greengrass コンポーネントを作成できます。

AWS IoT Greengrass でのインフラストラクチャセキュリティ

マネージドサービスである AWS IoT Greengrass は、Amazon Web Services のセキュリティプロセスの概要ホワイトペーパーに記載されている AWS グローバルネットワークセキュリティの手順で保護されています。

AWS 公開版 API コールを使用して、ネットワーク経由で AWS IoT Greengrass にアクセスします。クライアントで Transport Layer Security (TLS) 1.2 以降がサポートされている必要があります。TLS 1.3 以降が推奨されます。また、Ephemeral Diffie-Hellman (DHE) や Elliptic Curve Ephemeral Diffie-Hellman (ECDHE) などの Perfect Forward Secrecy (PFS) を使用した暗号スイートもクライアントでサポートされている必要があります。これらのモードは、Java 7 以降など、最近のほとんどのシステムでサポートされています。

リクエストは、アクセスキー ID と、IAM プリンシパルに関連付けられているシークレットのアクセスキーを使用して署名する必要があります。または、AWS Security Token Service (AWS STS) を使用して、一時的なセキュリティ認証情報を生成し、リクエストに署名することもできます。

AWS IoT Greengrass 環境では、デバイスは X.509 証明書と暗号化キーを使用して、AWS クラウドへの接続と認証を行います。詳細については、「the section called “デバイス認証と認可”」を参照してください。

AWS IoT Greengrass での設定と脆弱性の分析

IoT 環境は、多様な機能を持ち、存続期間が長く、地理的に分散される多数のデバイスで構成されることがあります。このような特性によってデバイスのセットアップが複雑になり、エラーを起こしやすくなります。また、デバイスの計算能力、メモリ、ストレージの機能には制約があるため、デバイス自体での暗号化や他の形式のセキュリティの使用が制限されます。さらに、デバイスは多く場合、既知の脆弱性を持つソフトウェアを使用しています。このような要素が原因で、IoT デバイスはハッカーの魅力的なターゲットとなり、継続的に保護することが困難になっています。

AWS IoT Device Defender は、セキュリティ上の問題とベストプラクティスからの逸脱を特定するツールを用意することでこれらの課題に対処します。AWS IoT Device Defender を使用すると、接続されたデバイスを分析、監査、監視して異常な動作を検出し、セキュリティリスクを軽減できます。AWS IoT Device Defender は、デバイスを監査し、セキュリティのベストプラクティスに準拠していることを確認して、デバイスでの異常な動作を検出できます。これにより、デバイス間でセキュリティポリシーを維持し、デバイスが侵害された場合にはすばやく応答することができます。詳細については、次のトピックを参照してください。

- Device Defender コンポーネント
- AWS IoT Core デベロッパーガイドの AWS IoT Device Defender。

AWS IoT Greengrass 環境では、次の考慮事項に注意する必要があります。

- 物理デバイス、デバイスのファイルシステム、ローカルネットワークの保護はお客様の責任です。
- AWS IoT Greengrass は、Greengrass コンテナで実行されているかどうかに関わらず、ユーザー定義の Greengrass コンポーネントのネットワーク分離を強制しません。したがって、Greengrass コンポーネントはシステム内またはネットワークを介して、外部で実行されている他のプロセスと通信することができます。

AWS IoT Greengrass V2 におけるコードの整合性

AWS IoT Greengrass は、AWS クラウド からのソフトウェアコンポーネントを AWS IoT Greengrass Coreコアソフトウェアを実行するデバイスにデプロイします。これらのソフトウェアコ

ンポーネントには、AWS から提供されるコンポーネントと、ユーザーが AWS アカウント にアップロードするカスタムコンポーネントが含まれます。すべてのコンポーネントは、レシピで構成されます。レシピは、コンポーネントのメタデータと任意の数のアーティファクトを定義します。アーティファクトとは、コンパイルされたコードや静的リソースなどのコンポーネントバイナリです。コンポーネントアーティファクトは Amazon S3 に保存されます。

Greengrass コンポーネントを開発しデプロイする際には、AWS アカウント 内とデバイス上にあるコンポーネントアーティファクトを扱った、次の簡単な手順に従ってください。

1. アーティファクトを作成し、S3 バケットにアップロードします。
2. AWS IoT Greengrass サービスのレシピとアーティファクトからコンポーネントを作成します。これにより、各アーティファクトの暗号ハッシュが計算されます。
3. Greengrass コアデバイスにコンポーネントをデプロイします。これにより Greengrass コアデバイスが各アーティファクトをダウンロードし、整合性を検証します。

AWS は、S3 バケットにアーティファクトをアップロードした後に、アーティファクトの整合性を維持する責任を担っており、これには Greengrass コアデバイスにコンポーネントをデプロイする場合も含まれます。S3 バケットにアーティファクトをアップロードする前は、ソフトウェアのアーティファクトを保護する責任はユーザーにあります。また、ユーザーには、AWS アカウント 内のリソースへのアクセスを保護する責任もあります。これには、コンポーネントアーティファクトをアップロードする S3 バケットを含まれます。

> **Note**
>
> Amazon S3 では S3 オブジェクトロックと呼ばれる機能が提供されており、ユーザーはこれを利用して、AWS アカウント の S3 バケット内のコンポーネントアーティファクトに対する変更から保護することができます。S3 オブジェクトロックを使用することで、コンポーネントのアーティファクトが削除または上書きされるのを防止することができます。詳細については、「Amazon Simple Storage Service ユーザーガイド」の「S3 オブジェクトロックを使用する」を参照してください。

AWS がパブリックコンポーネントをパブリッシュするとき、およびカスタムコンポーネントをアップロードするときには、AWS IoT Greengrass が各コンポーネントアーティファクトに対する暗号ダイジェストを計算します。AWS IoT Greengrass は、各アーティファクトのダイジェストとそのダイジェストを計算するために使用されるハッシュアルゴリズムが含まれるように、コンポーネントレシピを更新します。このダイジェストにより、成果物の完全性が保証される。なぜなら、もし

アーティファクトが AWS クラウド 内やダウンロード中に変更されると、そのファイルダイジェストは、AWS IoT Greengrass がコンポーネントレシピに保存するダイジェストと一致しなくなるからです。詳細については、「コンポーネントレシピリファレンスのアーティファクト」を参照してください。

コアデバイスにコンポーネントをデプロイすると、AWS IoT Greengrass コアソフトウェアが、コンポーネントレシピと、レシピが定義するコンポーネントアーティファクトをダウンロードします。AWS IoT Greengrass コアソフトウェアは、ダウンロードされた各アーティファクトファイルのダイジェストを計算し、レシピ内のアーティファクトのダイジェストと比較します。ダイジェストが一致しない場合、デプロイは失敗となり、AWS IoT Greengrass コアソフトウェアは、ダウンロードしたアーティファクトをデバイスのファイルシステムから削除します。コアデバイスと AWS IoT Greengrass 間の接続がどのように保護されているかの詳細については、「転送中の暗号化」を参照してください。

コアデバイスのファイルシステム上にあるコンポーネントアーティファクトファイルを保護する責任は、ユーザーが担います。AWS IoT Greengrass コアソフトウェアは、アーティファクトを Greengrass ルートフォルダにある `packages` フォルダに保存します。AWS IoT Device Defender を使用して、コアデバイスの分析、監査および監視を実行することができます。詳細については、「AWS IoT Greengrass での設定と脆弱性の分析」を参照してください。

AWS IoT Greengrass とインターフェース VPC エンドポイント (AWS PrivateLink)

インターフェイス VPC エンドポイントを作成することにより、VPC と AWS IoT Greengrass コントロールプレーンの間でプライベート接続を確立できます。このエンドポイントを使用して、のコンポーネント、デプロイ、AWS IoT Greengrass サービスのコアデバイスを管理できます。インターフェイスエンドポイントは、AWS PrivateLink を利用しており、インターネットゲートウェイ、NAT デバイス、VPN 接続、AWS Direct Connect 接続のいずれを使用せずに、AWS IoT Greengrass API にプライベートアクセスを可能にする技術です。VPC のインスタンスは、パブリック IP アドレスがなくても AWS IoT Greengrass API と通信できます。VPC と AWS IoT Greengrass 間のトラフィックは、Amazon ネットワークを離れません。

各インターフェイスエンドポイントは、サブネット内の 1 つ、または複数の Elastic Network Interface によって表されます。

詳細については、Amazon VPC ユーザーガイドの「インターフェイス VPC エンドポイント (AWS PrivateLink) 」を参照してください。

トピック

AWS IoT Greengrass VPC エンドポイントに関する考慮事項

AWS IoT Greengrass のインターフェイス VPC エンドポイントをセットアップする前、「Amazon VPC ユーザーガイド」の「インターフェイスエンドポイントのプロパティと制限」を確認してください。さらに、次の点について注意してください:

- AWS IoT Greengrass は、VPC から実行されるすべてのコントロールプレーン API アクションの呼び出しをサポートしています。コントロールプレーンには、 CreateDeploymentや などのオペレーションが含まれますListEffectiveDeployments。コントロールプレーンには、データプレーンオペレーションである ResolveComponentCandidatesや Discover などのオペレーションは含まれません。
- AWS IoT Greengrass 用 VPC エンドポイントは、現在 AWS 中国リージョンでサポートされていません。

AWS IoT Greengrass コントロールプレーン操作用インターフェイス VPC エンドポイントの作成

Amazon VPC コンソールまたは AWS Command Line Interface (AWS CLI) を使用して、AWS IoT Greengrass コントロールプレーン用 VPC エンドポイントを作成できます。詳細については、「Amazon VPC ユーザーガイド」の「インターフェイスエンドポイントの作成」を参照してください。

AWS IoT Greengrass 用の VPC エンドポイントは、以下のサービス名を使用して作成します。

- com.amazonaws.*region*.greengrass

エンドポイントのプライベート DNS を有効にすると、リージョンのデフォルト DNS 名 (AWS IoT Greengrass など) を使用して、`greengrass.us-east-1.amazonaws.com` への API リクエストを実行できます。プライベート DNS はデフォルトで有効になっています。

詳細については、「Amazon VPC ユーザーガイド」の「インターフェイスエンドポイントを介したサービスへのアクセス」を参照してください。

AWS IoT Greengrass 用の VPC エンドポイントポリシーの作成

VPC エンドポイントに、AWS IoT Greengrass コントロールプレーン操作へのアクセスを制御するエンドポイントポリシーをアタッチできます。このポリシーでは、以下の情報を指定します。

- アクションを実行できるプリンシパル。
- プリンシパルが実行できるアクション。
- プリンシパルがアクションを実行できるリソース。

詳細については、Amazon VPC ユーザーガイドの「VPC エンドポイントでサービスへのアクセスを制御する」を参照してください。

Example 例: AWS IoT Greengrass アクション用の VPC エンドポイントポリシー

以下は、AWS IoT Greengrass 用のエンドポイントポリシーの例です。エンドポイントにアタッチされると、このポリシーは、すべてのリソースですべてのプリンシパルに、リストされている AWS IoT Greengrass アクションへのアクセス権を付与します。

```
{
    "Statement": [
        {
            "Principal": "*",
            "Effect": "Allow",
            "Action": [
                "greengrass:CreateDeployment",
                "greengrass:ListEffectiveDeployments"
            ],
            "Resource": "*"
        }
    ]
}
```

VPC で AWS IoT Greengrassコアデバイスを運用する

Greengrass コアデバイスを運用し、パブリックインターネットアクセスのない VPC でデプロイを実行できます。少なくとも、対応する DNS エイリアスを使用して次の VPC エンドポイントを設定

する必要があります。VPC エンドポイントを作成して使用する方法の詳細については、「Amazon VPC ユーザーガイド」の「VPC エンドポイントの作成」を参照してください。

> **Note**
>
> DNS レコードを自動的に作成する VPC 機能は、 AWS IoT dataおよび AWS IoT 認証情報に対して無効になっています。これらのエンドポイントを接続するには、プライベート DNS レコードを手動で作成する必要があります。詳細については、「インターフェイスエンドポイントのプライベート DNS」を参照してください。AWS IoT Core VPC の制限の詳細については、「VPC エンドポイントの制限」を参照してください。

前提条件

- 手動プロビジョニング手順を使用して AWS IoT Greengrass Core ソフトウェアをインストールする必要があります。詳細については、「手動リソースプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールする」を参照してください。

制限事項

- VPC での Greengrass コアデバイスの運用は、中国リージョンおよび ではサポートされていませんAWS GovCloud (US) Regions。
- AWS IoT data およびAWS IoT認証情報プロバイダー VPC エンドポイントの制限の詳細については、「制限事項」を参照してください。

VPC で動作するように Greengrass コアデバイスをセットアップする

1. AWS アカウント に AWS IoT エンドポイントを入手して、後で使用するために保存してください。デバイスはこれらのエンドポイントを使用して AWS IoT に接続します。以下の操作を実行します。

   a. AWS アカウント の AWS IoT データエンドポイントを取得します。

   ```
   aws iot describe-endpoint --endpoint-type iot:Data-ATS
   ```

   要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "endpointAddress": "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
}
```

b. AWS IoT の AWS アカウント 認証情報エンドポイントを取得します。

```
aws iot describe-endpoint --endpoint-type iot:CredentialProvider
```

要求が正常に処理された場合、レスポンスは次の例のようになります。

```
{
  "endpointAddress": "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
}
```

2. AWS IoT data および AWS IoT認証情報エンドポイント用の Amazon VPC インターフェイスを作成します。

a. 左側のメニューの仮想プライベートクラウドで、VPC エンドポイントコンソールに移動し、エンドポイント を選択し、エンドポイント を作成します。

b. 「エンドポイントの作成」ページで、次の情報を指定します。

- [Service category] (サービスカテゴリ) には [AWS のサービスs] を選択します。
- [Service Name] (サービス名) については、キーワード iot を入力して検索します。表示されるiotサービスのリストで、エンドポイントを選択します。

  AWS IoT Core データプレーン用の VPC エンドポイントを作成する場合は、リージョン AWS IoT Coreのデータプレーン API エンドポイントを選択します。エンドポイントは com.amazonaws.*region*.iot.data の形式です。

  AWS IoT Core 認証情報プロバイダーの VPC エンドポイントを作成する場合は、リージョンのAWS IoT Core認証情報プロバイダーエンドポイントを選択します。エンドポイントは com.amazonaws.*region*.iot.credentials の形式です。

  > **Note**
  >
  > 中国リージョンAWS IoT Coreのデータプレーンのサービス名は、 の形式になりますcn.com.amazonaws.*region*.iot.data。AWS IoT Core 認証情報プロバ

イダーの VPC エンドポイントの作成は、中国リージョンではサポートされていません。

- [VPC] と [Subnets] (サブネット) には、エンドポイントを作成する VPC と、エンドポイントネットワークを作成するアベイラビリティーゾーン (AZ) を選択します。
- [Enable DNS name] (DNS 名を有効にする) で、[Enable for this endpoint] (このエンドポイントで有効にする) が選択されていないことを確認してください。AWS IoT Core データプレーンもAWS IoT Core認証情報プロバイダーも、まだプライベート DNS 名をサポートしていません。
- [Security group] (セキュリティグループ) には、エンドポイントネットワークインターフェイスに関連付けるセキュリティグループを選択します。
- オプションで、タグを追加または削除できます。タグとは名前と値のペアで、エンドポイントに関連付けるために使用します。

c. [Create Endpoint] (エンドポイントの作成) をクリックして、VPC エンドポイントを作成します。

3. AWS PrivateLink エンドポイントを作成すると、エンドポイントの詳細 タブに DNS 名のリストが表示されます。このセクションで作成したこれらの DNS 名のいずれかを使用して、プライベートホストゾーン を設定できます。
4. Amazon S3 エンドポイントを作成します。詳細については、「Amazon S3 の VPC エンドポイントを作成する」を参照してください。
5. AWSが提供する Greengrass コンポーネントを使用している場合は、追加のエンドポイントと設定が必要になる場合があります。エンドポイントの要件を表示するには、 AWSが提供するコンポーネントのリストからコンポーネントを選択し、「要件」セクションを参照してください。例えば、ログマネージャーコンポーネントの要件では、このコンポーネントがエンドポイント へのアウトバウンドリクエストを実行できることが推奨されていますlogs.*region*.amazonaws.com。

   独自のコンポーネントを使用している場合は、依存関係を確認し、追加のテストを実行して、追加のエンドポイントが必要かどうかを判断する必要がある場合があります。
6. Greengrass nucleus 設定では、 を に設定greengrassDataPlaneEndpointする必要がありますiotdata。詳細については、Greengrass nucleus 設定を参照してください。
7. us-east-1 リージョンにいる場合は、Greengrass s3EndpointType nucleus 設定REGIONALで 設定パラメータを に設定します。この機能は、Greengrass nucleus バージョン 2.11.3 以降で使用できます。

Example 例: コンポーネント設定

```
{
"aws.greengrass.Nucleus": {
   "configuration": {
      "awsRegion": "us-east-1",
      "iotCredEndpoint": "xxxxxx.credentials.iot.region.amazonaws.com",
      "iotDataEndpoint": "xxxxxx-ats.iot.region.amazonaws.com",
      "greengrassDataPlaneEndpoint": "iotdata",
      "s3EndpointType": "REGIONAL"
      ...
     }
   }
}
```

次の表は、対応するカスタムプライベート DNS エイリアスに関する情報です。

サービス	VPC エンドポイントサービス名	VPC エンドポイントタイプ	カスタムプライベート DNS エイリアス	メモ
AWS IoT data	com.amazonaws. *region*.iot.d	インターフェイス	*prefix*-ats.iot. *region*.a s.com	プライベート DNS レコードは、アカウントの AWS IoT dataエンドポイントと一致する必要がありますaws iot describe-endpoint

サービス	VPC エンドポイントサービス名	VPC エンドポイントタイプ	カスタムプライベート DNS エイリアス	メモ
				--endpoint-type iot:Data-ATS 。
AWS IoT 認証情報	com.amazonaws. *region*.iot.credentials	インターフェイス	*prefix*.credentials.iot.*region*.amazonaws.com	プライベート DNS レコードは、アカウントのAWS IoT認証情報エンドポイントと一致する必要がありますaws iot describe-endpoint --endpoint-type iot:CredentialProvider 。

サービス	VPC エンドポイントサービス名	VPC エンドポイントタイプ	カスタムプライベート DNS エイリアス	メモ
Amazon S3	`com.amazonaws.`*`region`*`.s3`	インターフェイス		DNS レコードが自動的に作成されます。

AWS IoT Greengrass のセキュリティのベストプラクティス

このトピックでは、AWS IoT Greengrass のセキュリティのベストプラクティスについて説明します。

最小限のアクセス許可の付与

コンポーネントに対する最小特権の原則に従うため、コンポーネントを非特権ユーザーとして実行します。どうしても必要な場合を除いて、コンポーネントはルートとして実行しないでください。

IAM ロールの最小アクセス許可セットを使用します。IAM ポリシーの `Action` プロパティおよび `Resource` プロパティに対する `*` ワイルドカードの使用を制限します。代わりに、可能な場合はアクションとリソースの有限セットを宣言します。最小特権およびその他のポリシーのベストプラクティスの詳細については、「the section called “ポリシーのベストプラクティス”」を参照してください。

最小特権のベストプラクティスは、Greengrass コアにアタッチする AWS IoT ポリシーにも適用されます。

Greengrass コンポーネントで認証情報をハードコードしない

ユーザー定義の Greengrass コンポーネントで認証情報をハードコードしないでください。認証情報をより適切に保護するには:

- AWS サービスとやり取りするには、[Greengrass core device service role] (Greengrass コアデバイスのサービスロール)で、特定のアクションとリソースに対するアクセス許可を定義します。

- [secret manager component] (シークレットマネージャーコンポーネント)を使用して認証情報を保存します。または、関数が AWS SDK を使用する場合は、デフォルトの認証情報プロバイダチェーンの認証情報を使用します。

機密情報を記録しない

認証情報やその他の個人を特定できる情報 (PII) のログを記録しないようにしてください。コアデバイスのローカルログへのアクセスにはルートアクセス許可が必要であり、CloudWatch Logs へのアクセスには IAM のアクセス許可が必要ですが、次の防止策を実施することをお勧めします。

- MQTT トピックパスに機密情報を使用しないでください。
- AWS IoT Core レジストリのデバイス (モノ) 名、種類、属性に機密情報を使用しないでください。
- ユーザー定義の Greengrass コンポーネントまたは Lambda 関数に機密情報を記録しないでください。
- Greengrass リソースの名前と ID に機密情報を使用しないでください。
  - コアデバイス
  - コンポーネント
  - デプロイ
  - Loggers

デバイスのクロックを同期させる

デバイスの時刻を正確に保つことが重要です。X.509 証明書には有効期限の日時があります。デバイスのクロックは、サーバー証明書が現在も有効であることを確認するために使用されます。デバイスのクロックは、時間の経過とともにドリフトしたり、バッテリが放電したりする可能性があります。

詳細については、「AWS IoT Core デベロッパーガイド」の「[Keep your device's clock in sync] (デバイスのクロックを同期させる)」ベストプラクティスを参照してください。

暗号スイートの推奨事項

Greengrass はデフォルトで、デバイスで使用可能な最新の TLS Cipher Suites を選択します。デバイス上のレガシー暗号スイートの使用を無効にすることを検討してください。たとえば、CBC 暗号スイートなどです。

詳細については、「Java 暗号化設定」を参照してください。。

以下も参照してください。

- 「AWS IoT デベロッパーガイド」の「AWS IoT Core でのセキュリティのベストプラクティス」
- 「AWS のモノのインターネット - 公式ブログ」に掲載の「産業における IoT ソリューションにおける 10 のセキュリティゴールデンルール」

AWS IoT Device Tester for AWS IoT Greengrass V2 を使用する

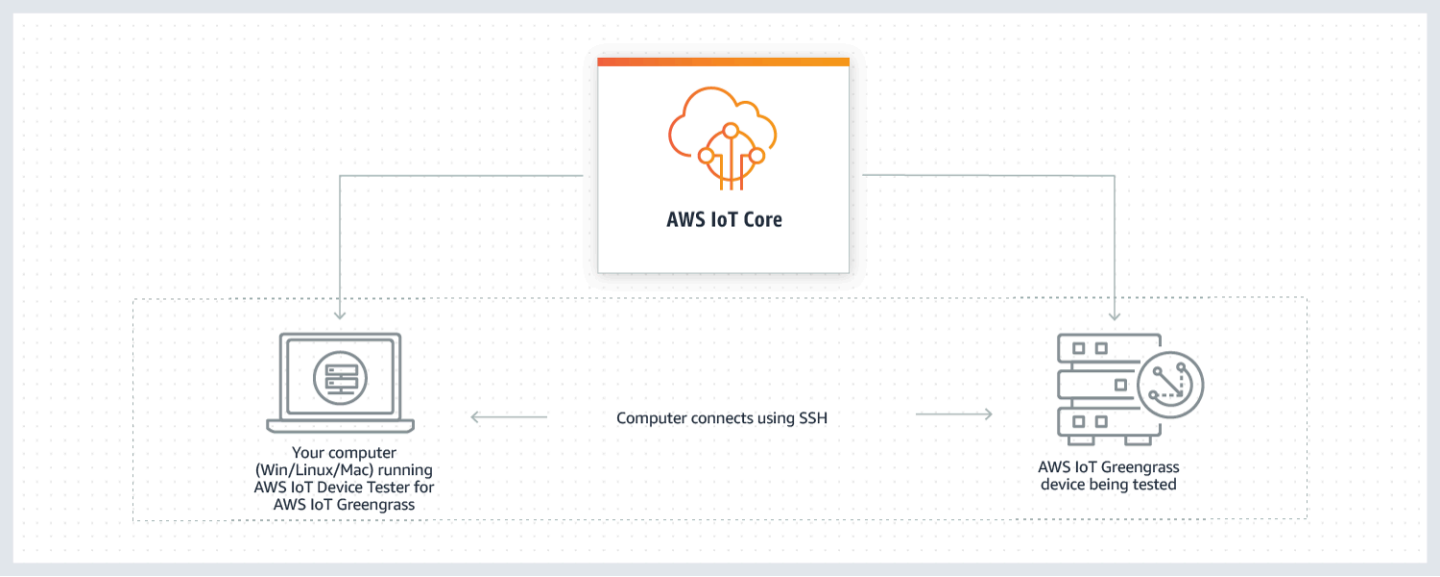
AWS IoT Device Tester (IDT) は、IoT デバイスを検証できる、ダウンロード可能なテストフレームワークです。IDT for AWS IoT Greengrass を使用して AWS IoT Greengrass 認定スイートを実行し、デバイス用のカスタムテストスイートを作成し実行することができます。

IDT for AWS IoT Greengrass は、テスト対象のデバイスに接続されているホストコンピュータ (Windows、Mac、または Linux) で動作します。また、テストを実行して結果を集計します。また、テストプロセスを管理するためのコマンドラインインターフェイスも用意されています。

AWS IoT Greengrass 認定スイート

AWS IoT Device Tester for AWS IoT Greengrass V2 を使用すると、AWS IoT Greengrass Core ソフトウェアがハードウェアで動作し、 AWS クラウド と通信できる状態であることを確認できます。また、 を使用して end-to-end テストも実行しますAWS IoT Core。たとえば、デバイスがコンポーネントをデプロイしてアップグレードできることを検証します。

AWS Partner Device Catalog にハードウェアを追加する場合は、AWS IoT Greengrass 認定スイートを実行して、AWS IoT に送信できるテストレポートを生成してください。詳細については、AWS デバイス認定プログラムを参照してください。



IDT for AWS IoT Greengrass V2 は、テストスイートとテストグループの概念を使用してテストを整理します。

- テストスイートは、デバイスが AWS IoT Greengrass の特定のバージョンで動作することを確認するために使用されるテストグループのセットです。
- テストグループは、コンポーネントデプロイなど、特定の機能に関連する個々のテストのセットです。

詳細については、「IDT を使用して AWS IoT Greengrass 認定スイートを実行する」を参照してください。

カスタムテストスイート

IDT v4.0.1 以降、IDT for AWS IoT Greengrass V2 では、標準化された構成設定および結果形式と、デバイスやデバイスソフトウェア用のカスタムテストスイートを開発できるテストスイート環境が統合されています。独自の内部検証用のカスタムテストを追加したり、デバイス検証のためにこれらのテストを顧客に提供したりできます。

テスト作成者がカスタムテストスイートをどのように設定するかによって、カスタムテストスイートの実行に必要な設定が変わってきます。詳細については、「IDT を使用して独自のテストスイートを開発および実行する」を参照してください。

AWS IoT Device Tester for AWS IoT Greengrass V2 のサポートされているバージョン

このトピックでは、IDT for AWS IoT Greengrass V2 のサポートされているバージョンを示します。ベストプラクティスとして、ターゲットバージョンの AWS IoT Greengrass V2 をサポートする最新バージョンの IDT for AWS IoT Greengrass V2 を使用することをお勧めします。の新しいリリースでは、IDT for AWS IoT Greengrass V2 の新しいバージョンをダウンロードする必要がある AWS IoT Greengrass 場合があります。IDT for AWS IoT Greengrass V2 AWS IoT Greengrass が使用しているのバージョンと互換性がない場合、テスト実行を開始すると通知を受け取ります。

ソフトウェアをダウンロードすると、AWS IoT Device Tester ライセンス契約に同意したと見なされます。

> **Note**
>
> 複数のユーザーが NFS ディレクトリや Windows ネットワーク共有フォルダなどの共有場所から IDT を実行することはお勧めしません。IDT パッケージをローカルドライブに展開し、ローカルワークステーションで IDT バイナリを実行することをお勧めします。

for AWS IoT Greengrass V2 の IDT の最新バージョン

このバージョンの IDT for AWS IoT Greengrass V2 は、ここに記載されている AWS IoT Greengrass バージョンで使用できます。

IDT v4.9.2 for AWS IoT Greengrass

サポートされている AWS IoT Greengrass バージョン：

- Greengrass nucleus v2.12.0、v2.11.0、v2.10.0、v2.9.5

IDT ソフトウェアダウンロード:

- IDT v4.9.2 と Linux 用テストスイート GGV2Q_2.5.2
- IDT v4.9.2 とテストスイート GGV2Q_2.5.2 for macOS
- IDT v4.9.2 とテストスイート GGV2Q_2.5.2 for Windows

リリースノート:

- Java 8 が廃止されたために Lambda テストスイートが失敗する問題を修正しました。

追加のメモ：

- デバイスが HSM を使用していて、nucleus 2.10.x を使用している場合は、Greengrass nucleus バージョン 2.11.0 以降に移行します。

テストスイートのバージョン:

`GGV2Q_2.5.2`

- 2024.03.18 をリリース

AWS IoT Device Tester for AWS IoT Greengrass V2 のサポートされていないバージョン

このトピックでは、サポートされていないバージョンの IDT for AWS IoT Greengrass V2 を一覧表示します。サポートされていないバージョンのバグ修正や更新プログラムは受けられません。詳細については、「the section called “AWS IoT Device Tester の のサポートポリシー AWS IoT Greengrass”」を参照してください。

の IDT v4.9.1 AWS IoT Greengrass

リリースノート:

- AWS IoT Greengrass Core ソフトウェアバージョン 2.12.0、2.11.0、2.10.0、および 2.9.5 を実行しているデバイスを検証および認定できます。

- 軽微なバグを修正。

テストスイートのバージョン:

`GGV2Q_2.5.1`

- 2023.10.05 をリリース

の IDT v4.7.0 AWS IoT Greengrass

サポートされている AWS IoT Greengrass バージョン :

- Greengrass nucleus v2.11.0、v2.10.0、v2.9.5

リリースノート:

- AWS IoT Greengrass Core ソフトウェアバージョン 2.11.0、2.10.0、および 2.9.5 を実行しているデバイスを検証および認定できます。
- IDT ユーザーデータの値を AWS Systems Manager Parameter Store に保存し、プレースホルダー構文を使用して設定にフェッチするためのサポートを追加しました。
- 軽微なバグを修正。

テストスイートのバージョン:

`GGV2Q_2.5.0`

- リリース日: 2022 年 12 月 13 日

の IDT v4.5.11 AWS IoT Greengrass

リリースノート:

- AWS IoT Greengrass Core ソフトウェアバージョン 2.9.1、2.9.0、2.8.1、2.8.0、2.7.0、2.6.0 を実行しているデバイスを、検証および認定できるようになりました。
- コアデバイスで PreInstalled Greengrass をテストするサポートを追加しました。
- 軽微なバグを修正。

テストスイートのバージョン:

`GGV2Q_2.4.1`

- リリース日: 2022 年 10 月 13 日

IDT v4.5.8 for AWS IoT Greengrass

リリースノート:

- AWS IoT Greengrass Core ソフトウェアバージョン 2.7.0、2.6.0、および 2.5.6 を実行しているデバイスを検証および認定できます。
- コアデバイスで PreInstalled Greengrass を使用してテストできます。

- 軽微なバグを修正。

テストスイートのバージョン:

`GGV2Q_2.4.0`

- リリース日: 2022 年 8 月 12 日

IDT v4.5.3 for AWS IoT Greengrass

リリースノート:

- AWS IoT Greengrass Core ソフトウェアバージョン 2.7.0、2.6.0、2.5.6、2.5.5、2.5.4、2.5.3 を実行しているデバイスを検証および認定できます。
- ECR ベースの Docker イメージを使用するための更新 DockerApplicationManager テスト。
- 軽微なバグを修正。

テストスイートのバージョン:

`GGV2Q_2.3.1`

- リリース日: 2022 年 4 月 15 日

の IDT v4.5.1 AWS IoT Greengrass

リリースノート:

- AWS IoT Greengrass Core ソフトウェア v2.5.3 を実行しているデバイスを検証し認定できるようになりました。
- ハードウェアセキュリティモジュール (HSM) を使用して、 AWS IoT Greengrass Core ソフトウェアで使用される秘密キーと証明書を格納する Linux ベースのデバイスの検証と適合に対するサポートを追加しました。
- カスタムテストスイートを設定するための新しい IDT テストオーケストレーターを実装しました。詳細については、「IDT テストオーケストレーターを設定する」を参照してください。
- 軽微なバグを追加で修正しました。

テストスイートのバージョン:

`GGV2Q_2.3.0`

- リリース日: 2022 年 1 月 11 日

の IDT v4.4.1 AWS IoT Greengrass

リリースノート:

- AWS IoT Greengrass Core ソフトウェア v2.5.2 を実行しているデバイスを検証および認定できます。

- テスト対象のデバイスが AWS リソースとやり取りするために引き受けるトークン交換ロールとしてユーザー定義の IAM ロールを使用するためのサポートを追加しました。

  IAM ロールは、`userdata.json` ファイルで指定できます。カスタムロールを指定すると、IDT はテストの実行中にデフォルトのトークン交換ロールを作成する代わりに、このカスタム IAM ロールを使用します。
- 軽微なバグを追加で修正しました。

テストスイートのバージョン:

`GGV2Q_2.2.1`

- リリース日: 2021 年 12 月 12 日

IDT v4.4.0 for AWS IoT Greengrass

リリースノート:

- AWS IoT Greengrass Core ソフトウェア v2.5.0 を実行しているデバイスを検証および認定できます。
- Windows で AWS IoT Greengrass Core ソフトウェアを実行するデバイスの検証と認定のサポートを追加しました。
- Secure Shell (SSH) デバイス接続の公開キー検証の使用をサポートするようになりました。
- IDT アクセス許可の IAM ポリシーを、セキュリティのベストプラクティスで向上しました。
- 軽微なバグを追加で修正しました。

テストスイートのバージョン:

`GGV2Q_2.1.0`

- リリース日: 2021 年 11 月 19 日

の IDT v4.2.0 AWS IoT Greengrass

リリースノート:

- AWS IoT Greengrass Core ソフトウェア v2.2.0 以降のバージョンを実行しているデバイスでの以下の機能の認定のサポートが含まれています。
  - Docker - デバイスが Amazon Elastic Container Registry (Amazon ECR) から Docker コンテナイメージをダウンロードできることを検証します。
  - 機械学習 - 深層学習ランタイムまたは TensorFlow Lite ML フレームワークを使用して、デバイスが機械学習 (ML) 推論を実行できることを検証します。
  - ストリームマネージャー - デバイスが AWS IoT Greengrass ストリームマネージャーをダウンロード、インストール、実行できることを検証します。

- AWS IoT Greengrass Core ソフトウェア v2.4.0、v2.3.0、v2.2.0、v2.1.0 を実行しているデバイスを検証および認定できます。
- 各テストケースのテストログを *<device-tester-extract-location>*/results/*<execution-id>*/logs/*<test-group-id>* ディレクトリ内の個別の *#test-case-id#* フォルダにグループ化します。
- 軽微なバグを追加で修正しました。

テストスイートのバージョン:

GGV2Q_2.0.1

- リリース日: 2021 年 8 月 31 日

の IDT v4.1.0 AWS IoT Greengrass

リリースノート:

- AWS IoT Greengrass Core ソフトウェア v2.3.0、v2.2.0、v2.1.0、v2.0.5 を実行しているデバイスを検証および認定できます。
- GreengrassNucleusVersion と GreengrassCLIVersion プロパティを指定する要件が排除され、userdata.json 設定が向上しました。
- AWS IoT Greengrass Core ソフトウェア v2.1.0 以降のバージョンの Lambda および MQTT 機能認定のサポートが含まれています。IDT for AWS IoT Greengrass V2 を使用して、コアデバイスが Lambda 関数を実行できること、およびデバイスが AWS IoT Core MQTT トピックを発行およびサブスクライブできることを検証できるようになりました。
- ロギング機能が向上しました。
- 軽微なバグを追加で修正しました。

テストスイートのバージョン:

GGV2Q_1.1.1

- リリース日: 2021 年 6 月 18 日

の IDT v4.0.2 AWS IoT Greengrass

リリースノート:

- AWS IoT Greengrass Core ソフトウェア v2.1.0 を実行しているデバイスを検証および認定できます。
- AWS IoT Greengrass Core ソフトウェア v2.1.0 以降のバージョンの Lambda および MQTT 機能認定のサポートを追加しました。IDT for AWS IoT Greengrass V2 を使用して、コアデバイスが Lambda 関数を実行できること、およびデバイスが AWS IoT Core MQTT トピックを発行およびサブスクライブできることを検証できるようになりました。

- ロギング機能が向上しました。
- 軽微なバグを追加で修正しました。

テストスイートのバージョン:

`GGV2Q_1.1.1`

- リリース日: 2021 年 5 月 5 日

の IDT v4.0.1 AWS IoT Greengrass

リリースノート:

- AWS IoT Greengrass バージョン 2 ソフトウェアを実行しているデバイスを検証および認定できます。
- AWS IoT Device Tester for AWS IoT Greengrassを使用して、カスタムテストスイートを開発および実行できるようになりました。詳細については、「IDT を使用して独自のテストスイートを開発および実行する」を参照してください。
- macOS および Windows 用のコード署名付き IDT アプリケーションを提供します。macOS では、IDT のセキュリティ例外を許可する必要がある場合があります。詳細については、「macOS でのセキュリティ例外」を参照してください。

テストスイートのバージョン:

`GGV2Q_1.0.0`

- リリース日: 2020 年 12 月 22 日
- テストスイートは、`features` 配列の対応する `value` を `yes` に設定しない限り、認定に必要なテストのみを実行します。

IDT for AWS IoT Greengrass V2 のダウンロード

このトピックでは、 AWS IoT Device Tester for AWS IoT Greengrass V2 をダウンロードするオプションについて説明します。次のいずれかのソフトウェアダウンロードリンクを使用するか、指示に従ってプログラムで IDT をダウンロードできます。

トピック

- IDT を手動でダウンロードする
- IDT をプログラムでダウンロード

ソフトウェアをダウンロードすると、AWS IoT Device Tester ライセンス契約に同意したと見なされます。

> **Note**
>
> 複数のユーザーが NFS ディレクトリや Windows ネットワーク共有フォルダなどの共有場所から IDT を実行することはお勧めしません。IDT パッケージをローカルドライブに展開し、ローカルワークステーションで IDT バイナリを実行することをお勧めします。

IDT を手動でダウンロードする

このトピックでは、IDT for AWS IoT Greengrass V2 のサポートされているバージョンを示します。ベストプラクティスとして、ターゲットバージョンの AWS IoT Greengrass V2 をサポートする最新バージョンの IDT for AWS IoT Greengrass V2 を使用することをお勧めします。の新しいリリースでは、IDT for AWS IoT Greengrass V2 の新しいバージョンをダウンロードする必要がある AWS IoT Greengrass 場合があります。IDT for AWS IoT Greengrass V2 が使用している AWS IoT Greengrass のバージョンと互換性がない場合、テスト実行を開始すると通知を受け取ります。

の IDT v4.9.2 AWS IoT Greengrass

サポートされている AWS IoT Greengrass バージョン：

- Greengrass nucleus v2.12.0、v2.11.0、v2.10.0、v2.9.5

IDT ソフトウェアダウンロード:

- IDT v4.9.2 と Linux 用テストスイート GGV2Q_2.5.2
- IDT v4.9.2 と macOS 用テストスイート GGV2Q_2.5.2
- テストスイート GGV2Q_2.5.2 for Windows を使用した IDT v4.9.2

リリースノート:

- Java 8 が廃止されたために Lambda テストスイートが失敗する問題を修正しました。

追加のメモ：

- デバイスが HSM を使用していて、nucleus 2.10.x を使用している場合は、Greengrass nucleus バージョン 2.11.0 以降に移行します。

テストスイートのバージョン:

`GGV2Q_2.5.2`

- 2024.03.18 をリリース

IDT をプログラムでダウンロード

IDT には、プログラムで IDT をダウンロードできる URL の取得に使用できる API オペレーションが用意されています。この API オペレーションを使用して、IDT が最新バージョンであることを確認することもできます。この API オペレーションには、以下のエンドポイントがあります。

```
https://download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt
```

この API オペレーションを呼び出すには、**iot-device-tester:LatestIdt** アクションを実行するための権限が必要です。 AWS 署名を含めて、サービス名iot-device-testerとして を使用します。

API リクエスト

HostOs – ホストマシンのオペレーティングシステム。次のオプションから選択します。

- `mac`
- `linux`
- `windows`

TestSuiteType – テストスイートのタイプ。次のオプションを選択します。

`GGV2` – IDT for AWS IoT Greengrass V2

ProductVersion

(オプション) Greengrass nucleus のバージョン。サービスは、Greengrass nucleus のそのバージョンと互換性のある最新バージョンの IDT を返します。このオプションを指定しない場合、サービスは最新バージョンの IDT を返します。

API レスポンス

API レスポンスの形式は次のとおりです。`DownloadURL` には zip ファイルが付属しています。

```
{
    "Success": True or False,
    "Message": Message,
    "LatestBk": {
        "Version": The version of the IDT binary,
        "TestSuiteVersion": The version of the test suite,
        "DownloadURL": The URL to download the IDT Bundle, valid for one hour
```

```
    }
}
```

例

プログラムで IDT をダウンロードするには、以下の例を参照してください。これらの例では、AWS_ACCESS_KEY_ID に保存した認証情報および AWS_SECRET_ACCESS_KEY 環境変数が使用されます。セキュリティのベストプラクティスに従い、認証情報はコードに保存しないでください。

Example 例: cURL バージョン 7.75.0 以降を使用したダウンロード (Mac および Linux)

cURL バージョン 7.75.0 以降の場合、aws-sigv4 フラグを使用して API リクエストに署名できます。この例では、レスポンスからのダウンロード URL の解析に jq を使用します。

> **Warning**
>
> aws-sigv4 フラグでは、curl GET リクエストのクエリパラメータが HostOs/ProductVersion/TestSuiteType または HostOs/TestSuiteType の順序である必要があります。注文が一致しない場合、API ゲートウェイから正規文字列の署名が一致しないというエラーが発生します。
> オプションの パラメータProductVersionが含まれている場合は、「 for AWS IoT Greengrass V2 のサポートされているバージョン AWS IoT Device Tester 」に記載されているように、サポートされている製品バージョンを使用する必要があります。

- *us-west-2* を に置き換えます AWS リージョン。リージョンコードの一覧については、「リージョンエンドポイント」を参照してください。
- *linux* をホストマシンのオペレーティングシステムに置き換えます。
- *2.5.3* を AWS IoT Greengrass nucleus のバージョンに置き換えます。

```
url=$(curl --request GET "https://
download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt?
HostOs=linux&ProductVersion=2.5.3&TestSuiteType=GGV2" \
--user $AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY \
--aws-sigv4 "aws:amz:us-west-2:iot-device-tester" \
| jq -r '.LatestBk["DownloadURL"]')

curl $url --output devicetester.zip
```

Example 例: 以前のバージョンの cURL を使用したダウンロード (Mac および Linux)

次の cURL コマンドは、署名して計算する AWS 署名で使用できます。署名に署名して計算する方法の詳細については AWS 、AWS 「 API リクエストの署名」を参照してください。

- *linux* をホストマシンのオペレーティングシステムに置き換えます。
- *#######*を **20220210T004606Z** などの日付と時刻に置き換えます。
- *##*を **20220210** などの日付に置き換えます。
- を *AWSRegion*に置き換えます AWS リージョン。リージョンコードの一覧については、「リージョンエンドポイント」を参照してください。
- を、生成したAWS 署名*AWSSignature*に置き換えます。

```
curl --location --request GET 'https://
download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt?
HostOs=linux&TestSuiteType=GGV2' \
--header 'X-Amz-Date: Timestamp \
--header 'Authorization: AWS4-HMAC-SHA256 Credential=$AWS_ACCESS_KEY_ID/Date/AWSRegion/
iot-device-tester/aws4_request, SignedHeaders=host;x-amz-date, Signature=AWSSignature'
```

Example 例: Python スクリプトを使用したダウンロード

この例では Python リクエストライブラリを使用しています。この例は、 AWS 全般のリファレンスの AWS API リクエストに署名する Python の例から適用されています。

- *us-west-2* を、ご利用のリージョンに置き換えます。リージョンコードの一覧については、「リージョンエンドポイント」を参照してください。
- *linux* をホストマシンのオペレーティングシステムに置き換えます。

```
# Copyright 2010-2022 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# This file is licensed under the Apache License, Version 2.0 (the "License").
# You may not use this file except in compliance with the License. A copy of the
#License is located at
#
# http://aws.amazon.com/apache2.0/
#
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS
# OF ANY KIND, either express or implied. See the License for the specific
```

```
# language governing permissions and limitations under the License.

# See: http://docs.aws.amazon.com/general/latest/gr/sigv4_signing.html
# This version makes a GET request and passes the signature
# in the Authorization header.
import sys, os, base64, datetime, hashlib, hmac
import requests # pip install requests
# ************* REQUEST VALUES *************
method = 'GET'
service = 'iot-device-tester'
host = 'download.devicetester.iotdevicesecosystem.amazonaws.com'
region = 'us-west-2'
endpoint = 'https://download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt'
request_parameters = 'HostOs=linux&TestSuiteType=GGV2'

# Key derivation functions. See:
# http://docs.aws.amazon.com/general/latest/gr/signature-v4-examples.html#signature-v4-
examples-python
def sign(key, msg):
    return hmac.new(key, msg.encode('utf-8'), hashlib.sha256).digest()

def getSignatureKey(key, dateStamp, regionName, serviceName):
    kDate = sign(('AWS4' + key).encode('utf-8'), dateStamp)
    kRegion = sign(kDate, regionName)
    kService = sign(kRegion, serviceName)
    kSigning = sign(kService, 'aws4_request')
    return kSigning

# Read AWS access key from env. variables or configuration file. Best practice is NOT
# to embed credentials in code.
access_key = os.environ.get('AWS_ACCESS_KEY_ID')
secret_key = os.environ.get('AWS_SECRET_ACCESS_KEY')
if access_key is None or secret_key is None:
    print('No access key is available.')
    sys.exit()

# Create a date for headers and the credential string
t = datetime.datetime.utcnow()
amzdate = t.strftime('%Y%m%dT%H%M%SZ')
datestamp = t.strftime('%Y%m%d') # Date w/o time, used in credential scope

# ************* TASK 1: CREATE A CANONICAL REQUEST *************
# http://docs.aws.amazon.com/general/latest/gr/sigv4-create-canonical-request.html
# Step 1 is to define the verb (GET, POST, etc.)--already done.
```

```
# Step 2: Create canonical URI--the part of the URI from domain to query
# string (use '/' if no path)
canonical_uri = '/latestidt'
# Step 3: Create the canonical query string. In this example (a GET request),
# request parameters are in the query string. Query string values must
# be URL-encoded (space=%20). The parameters must be sorted by name.
# For this example, the query string is pre-formatted in the request_parameters
 variable.
canonical_querystring = request_parameters
# Step 4: Create the canonical headers and signed headers. Header names
# must be trimmed and lowercase, and sorted in code point order from
# low to high. Note that there is a trailing \n.
canonical_headers = 'host:' + host + '\n' + 'x-amz-date:' + amzdate + '\n'
# Step 5: Create the list of signed headers. This lists the headers
# in the canonical_headers list, delimited with ";" and in alpha order.
# Note: The request can include any headers; canonical_headers and
# signed_headers lists those that you want to be included in the
# hash of the request. "Host" and "x-amz-date" are always required.
signed_headers = 'host;x-amz-date'
# Step 6: Create payload hash (hash of the request body content). For GET
# requests, the payload is an empty string ("").
payload_hash = hashlib.sha256(('').encode('utf-8')).hexdigest()
# Step 7: Combine elements to create canonical request
canonical_request = method + '\n' + canonical_uri + '\n' + canonical_querystring + '\n'
 + canonical_headers + '\n' + signed_headers + '\n' + payload_hash

# ************* TASK 2: CREATE THE STRING TO SIGN*************
# Match the algorithm to the hashing algorithm you use, either SHA-1 or
# SHA-256 (recommended)
algorithm = 'AWS4-HMAC-SHA256'
credential_scope = datestamp + '/' + region + '/' + service + '/' + 'aws4_request'
string_to_sign = algorithm + '\n' +  amzdate + '\n' +  credential_scope + '\n' +
 hashlib.sha256(canonical_request.encode('utf-8')).hexdigest()
# ************* TASK 3: CALCULATE THE SIGNATURE *************
# Create the signing key using the function defined above.
signing_key = getSignatureKey(secret_key, datestamp, region, service)
# Sign the string_to_sign using the signing_key
signature = hmac.new(signing_key, (string_to_sign).encode('utf-8'),
 hashlib.sha256).hexdigest()

# ************* TASK 4: ADD SIGNING INFORMATION TO THE REQUEST *************
# The signing information can be either in a query string value or in
# a header named Authorization. This code shows how to use a header.
# Create authorization header and add to request headers
```

```
authorization_header = algorithm + ' ' + 'Credential=' + access_key + '/' +
 credential_scope + ', ' +  'SignedHeaders=' + signed_headers + ', ' + 'Signature=' +
 signature
# The request can include any headers, but MUST include "host", "x-amz-date",
# and (for this scenario) "Authorization". "host" and "x-amz-date" must
# be included in the canonical_headers and signed_headers, as noted
# earlier. Order here is not significant.
# Python note: The 'host' header is added automatically by the Python 'requests'
 library.
headers = {'x-amz-date':amzdate, 'Authorization':authorization_header}

# ************* SEND THE REQUEST *************
request_url = endpoint + '?' + canonical_querystring
print('\nBEGIN REQUEST++++++++++++++++++++++++++++++++++++')
print('Request URL = ' + request_url)
response = requests.get(request_url, headers=headers)
print('\nRESPONSE++++++++++++++++++++++++++++++++++++')
print('Response code: %d\n' % response.status_code)
print(response.text)

download_url = response.json()["LatestBk"]["DownloadURL"]
r = requests.get(download_url)
open('devicetester.zip', 'wb').write(r.content)
```

IDT を使用して AWS IoT Greengrass 認定スイートを実行する

AWS IoT Device Tester for AWS IoT Greengrass V2 を使用して、 AWS IoT Greengrass Core ソフトウェアがハードウェア上で実行され、 と通信できることを確認できます AWS クラウド。また、 を使用して end-to-end テストも実行します AWS IoT Core。たとえば、デバイスがコンポーネントをデプロイしてアップグレードできることを検証します。

IDT for AWS IoT Greengrass V2 は、認定プロセスを容易に AWS アカウント するために、デバイスのテストに加えて、 に リソース (AWS IoT モノ、グループなど) を作成します。

これらのリソースを作成するために、IDT for AWS IoT Greengrass V2 は `config.json` ファイルで設定された AWS 認証情報を使用して、ユーザーに代わって API コールを実行します。これらのリソースは、テスト中にさまざまなタイミングでプロビジョニングされます。

IDT for AWS IoT Greengrass V2 を使用して AWS IoT Greengrass 認定スイートを実行すると、以下の手順が実行されます。

1. デバイスおよび認証情報の設定をロードして検証します。

2. 必要なローカルリソースとクラウドリソースを使用して選択したテストを実行します。
3. ローカルリソースとクラウドリソースをクリーンアップします。
4. ボードが資格に必要なテストに合格したかどうかを示すテストレポートを生成します。

テストスイートのバージョン

IDT for AWS IoT Greengrass V2 は、テストをテストスイートとテストグループに整理します。

- テストスイートは、デバイスが AWS IoT Greengrassの特定のバージョンで動作することを確認するために使用されるテストグループのセットです。
- テストグループは、コンポーネントデプロイなど、特定の機能に関連する個々のテストのセットです。

テストスイートは *major*.*minor*.*patch* 形式を使用してバージョン管理されます (例: `GGV2Q_1.0.0`)。IDT をダウンロードした際、パッケージには Greengrass 認定スイートの最新バージョンが含まれています。

> **Important**
>
> サポートされていないテストスイートのバージョンからのテストは、デバイスの認定には有効ではありません。IDT では、サポートされていないバージョンの認定レポートは印刷されません。詳細については、「the section called “AWS IoT Device Tester の のサポートポリシー AWS IoT Greengrass”」を参照してください。
> `list-supported-products` を実行して、現在のバージョンの IDT でサポートされている AWS IoT Greengrass およびテストスイートのバージョンを一覧表示できます。

テストグループの説明

コア資格に必要なテストグループ

これらのテストグループは、 AWS IoT Greengrass V2 デバイスを AWS Partner Device Catalog の認定に必要です。

Core 依存関係

デバイスが AWS IoT Greengrass Core ソフトウェアのすべてのソフトウェア要件とハードウェア要件を満たしていることを確認します。このテストグループには、以下のテストケースが含まれます。

Java バージョン

必要な Java バージョンがテスト対象のデバイスにインストールされていることを確認します。Java 8 以降 AWS IoT Greengrass が必要です。

PreTest 検証

デバイスがテストを実行するためのソフトウェア要件を満たしていることを確認します。

- Linux ベースのデバイスの場合、このテストでデバイスが以下の Linux コマンドを実行できることを確認します。

  `chmod`, `cp`, `echo`, `grep`, `kill`, `ln`, `mkinfo`, `ps`, `rm`, `sh`, `uname`
- Windows ベースのデバイスの場合、このテストでデバイスに以下の Microsoft ソフトウェアがインストールされていることを確認します。

  Powershell v5.1 以降、.NET v4.6.1 以降、Visual C++ 2017 以降、PsExecユーティリティ

バージョンチェッカー

AWS IoT Greengrass 提供された のバージョンが、使用している AWS IoT Device Tester のバージョンと互換性があることを確認します。

コンポーネント

デバイスがコンポーネントをデプロイしてアップグレードできることを検証します。このテストグループには、以下のテストが含まれます。

クラウドコンポーネント

クラウドコンポーネントに対するデバイスの能力を検証します。

ローカルコンポーネント

ローカルコンポーネントに対するデバイスの能力を検証します。

Lambda

このテストは Windows ベースのデバイスには適用されません。

デバイスが Java ランタイムを使用する Lambda 関数コンポーネントをデプロイできること、および Lambda 関数が AWS IoT Core MQTT トピックを作業メッセージのイベントソースとして使用できることを検証します。

MQTT

デバイスが AWS IoT Core MQTT トピックをサブスクライブして発行できることを検証します。

オプションのテストグループ

> **Note**
> このテストグループはオプションで、Linux ベースの Greengrass コアデバイスの認定にのみ使用されます。オプションのテストの資格を選択すると、デバイスが AWS Partner Device Catalog の追加機能とともに一覧表示されます。

Docker 依存関係

デバイスが、 AWSが提供する Docker アプリケーションマネージャー (`aws.greengrass.DockerApplicationManager`) コンポーネントを使用するために必要なすべての技術的依存関係を満たしていることを確認します。

Docker アプリケーションマネージャー認定

デバイスが Amazon ECR から Docker コンテナイメージをダウンロードできることを検証します。

機械学習の依存関係

デバイスが、 AWSが提供する機械学習 (ML) コンポーネントを使用するために必要なすべての技術的依存関係を満たしていることを確認します。

機械学習の推論テスト

深層学習ランタイムと TensorFlow Lite ML フレームワークを使用して、デバイスが ML 推論を実行できることを検証します。

ストリームマネージャーの依存関係

デバイスが AWS IoT Greengrass ストリームマネージャーをダウンロード、インストール、および実行できることを検証します。

ハードウェアセキュリティ統合 (HSI)

> **Note**
>
> このテストはIDT v4.5.1 以降で、Linux ベースのデバイスのみ利用できます。 AWS IoT Greengrass は現在、Windows デバイス用ハードウェアセキュリティ統合はサポートしていません。

ハードウェアセキュリティモジュール (HSM) に保存されているプライベートキーと証明書を使用して、デバイスが AWS IoT および AWS IoT Greengrass サービスへの接続を認証できることを検証します。このテストでは、 が提供する PKCS#11 AWSプロバイダコンポーネントが、ベンダーが提供する PKCS#11 ライブラリを使用して HSM とインターフェイスできることも検証します。 ???詳細については、「ハードウェアセキュリティ統合」を参照してください。

AWS IoT Greengrass 認定スイートを実行するための前提条件

このセクションでは、AWS IoT Device Tester (IDT) for AWS IoT Greengrass を使用するための前提条件について説明します。

最新バージョンの AWS IoT Device Tester for AWS IoT Greengrass のダウンロード

IDT の最新バージョンをダウンロードし、読み取り/書き込みアクセス許可を持つファイルシステム上の場所にソフトウェアを抽出します (*#device-tester-extract-location#*)。

> **Note**
>
> 複数のユーザーが NFS ディレクトリや Windows ネットワーク共有フォルダなどの共有場所から IDT を実行することはお勧めしません。IDT パッケージをローカルドライブに展開し、ローカルワークステーションで IDT バイナリを実行することをお勧めします。
> Windows では、パスの長さは 260 文字に制限されています。Windows を使用している場合は、パスが 260 文字以内になるようにして、IDT をルートディレクトリ (`C:\` または `D:\` など) に展開します。

AWS IoT Greengrass ソフトウェアをダウンロードする

IDT for AWS IoT Greengrass V2 は、デバイスが特定バージョンの AWS IoT Greengrass と互換性があるかどうかをテストします。次のコマンドを実行して、AWS IoT Greengrass Core ソフトウェアを `aws.greengrass.nucleus.zip` という名のファイルにダウンロードします。*`version`* を IDT バージョンのサポートされている nucleus コンポーネントのバージョンに置き換えます。

Linux or Unix

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip >
 aws.greengrass.nucleus.zip
```

Windows Command Prompt (CMD)

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip >
 aws.greengrass.nucleus.zip
```

PowerShell

```
iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip -
OutFile aws.greengrass.nucleus.zip
```

ダウンロードした `aws.greengrass.nucleus.zip` ファイルを *`<device-tester-extract-location>`*`/products/` フォルダに配置します。

> **Note**
>
> 同じオペレーティングシステムとアーキテクチャのこのディレクトリに複数のファイルを配置しないでください。

AWS アカウント を作成して設定する

AWS IoT Device Tester for AWS IoT Greengrass V2 を使用する前に、次のステップを実行する必要があります。

1. AWS アカウント をセットアップします。AWS アカウント を既にお持ちの場合は、ステップ 2 に進んでください。

2. IDT 用のアクセス許可を設定する。

このアカウントのアクセス許可により、IDT から AWS サービスにアクセスし、ユーザーに代わって AWS リソース (AWS IoT モノ、AWS IoT Greengrass コンポーネントなど) を作成できるようになります。

これらのリソースを作成するために、IDT for AWS IoT Greengrass V2 は `config.json` ファイルに設定されている AWS 認証情報を使用して、ユーザーに代わって API コールを行います。これらのリソースは、テスト中にさまざまなタイミングでプロビジョニングされます。

> **Note**
>
> ほとんどのテストは AWS 無料利用枠の対象となりますが、AWS アカウント アカウントにサインアップするときにクレジットカードを提供する必要があります。詳細については、「アカウントが無料利用枠の対象であるのに、支払い方法が必要なのはなぜですか?」を参照してください。

ステップ 1: AWS アカウント を設定する

このステップでは、AWS アカウント を作成して設定します。AWS アカウント を既にお持ちの場合は、the section called “ステップ 2: IDT 用のアクセス許可を設定する” に進んでください。

AWS アカウント をお持ちでない場合は、以下の手順を実行してアカウントを作成してください。

AWS アカウント にサインアップするには

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、通話呼び出しを受け取り、電話のキーパッドを使用して検証コードを入力するように求められます。

   AWS アカウントにサインアップすると、AWS アカウントのルートユーザーが作成されます。ルートユーザーには、アカウントのすべての AWS のサービスとリソースへのアクセス権があります。セキュリティのベストプラクティスとして、管理ユーザーに管理アクセスを割り当て、ルートユーザーアクセスが必要なタスクを実行する場合にのみ、ルートユーザーを使用してください。

管理者ユーザーを作成するには、以下のいずれかのオプションを選択します。

管理者を管理する方法を 1 つ選択します	目的	方法	以下の操作も可能
IAM Identity Center 内 (推奨)	短期の認証情報を使用して AWS にアクセスします。 これはセキュリティのベストプラクティスと一致しています。ベストプラクティスの詳細については、IAM ユーザーガイドの「IAM でのセキュリティのベストプラクティス」を参照してください。	AWS IAM Identity Center ユーザーガイドの「開始方法」の手順に従います。	AWS Command Line Interface ユーザーガイドの「AWS IAM Identity Center を使用するための AWS CLI の設定」に従って、プログラムによるアクセスを設定します。
IAM 内 (非推奨)	長期認証情報を使用して AWS にアクセスする。	IAM ユーザーガイドの「最初の IAM 管理者のユーザーおよびグループの作成」の手順に従います。	IAM ユーザーガイドの「IAM ユーザーのアクセスキーの管理」に従って、プログラムによるアクセスを設定します。

ステップ 2: IDT 用のアクセス許可を設定する

このステップでは、IDT for AWS IoT Greengrass V2 がテストを実行して IDT 使用状況データを収集するために使用するアクセス許可を設定します。AWS Management Consoleまたは AWS Command Line Interface (AWS CLI) を使用して、IDT 用の IAM ポリシーとテストユーザーを作成し、そのユーザーにポリシーをアタッチできます。IDT 用のテストユーザーをすでに作成している場合は、IDT テストを実行するようにデバイスを設定する に進みます。

IDT 用のアクセス許可を設定するには (コンソール)

1. IAM コンソールにサインインします。
2. 特定のアクセス許可を持つロールを作成するためのアクセス許可を付与するカスタマー管理ポリシーを作成します。

   a. ナビゲーションペインで ポリシーを選択してから ポリシーの作成 を選択します。

   b. を使用していない場合は PreInstalled、JSON タブでプレースホルダーコンテンツを次のポリシーに置き換えます。を使用している場合は PreInstalled、次のステップに進みます。

```
{
    "Version":"2012-10-17",
    "Statement":[
    {
      "Sid":"passRoleForResources",
      "Effect":"Allow",
      "Action":"iam:PassRole",
      "Resource":"arn:aws:iam::*:role/idt-*",
      "Condition":{
        "StringEquals":{
          "iam:PassedToService":[
            "iot.amazonaws.com",
            "lambda.amazonaws.com",
            "greengrass.amazonaws.com"
          ]
        }
      }
    },
    {
      "Sid":"lambdaResources",
      "Effect":"Allow",
      "Action":[
        "lambda:CreateFunction",
        "lambda:PublishVersion",
        "lambda:DeleteFunction",
        "lambda:GetFunction"
      ],
      "Resource":[
        "arn:aws:lambda:*:*:function:idt-*"
      ]
    },
    {
```

```
      "Sid":"iotResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateThing",
        "iot:DeleteThing",
        "iot:DescribeThing",
        "iot:CreateThingGroup",
        "iot:DeleteThingGroup",
        "iot:DescribeThingGroup",
        "iot:AddThingToThingGroup",
        "iot:RemoveThingFromThingGroup",
        "iot:AttachThingPrincipal",
        "iot:DetachThingPrincipal",
        "iot:UpdateCertificate",
        "iot:DeleteCertificate",
        "iot:CreatePolicy",
        "iot:AttachPolicy",
        "iot:DetachPolicy",
        "iot:DeletePolicy",
        "iot:GetPolicy",
        "iot:Publish",
        "iot:TagResource",
        "iot:ListThingPrincipals",
        "iot:ListAttachedPolicies",
        "iot:ListTargetsForPolicy",
        "iot:ListThingGroupsForThing",
        "iot:ListThingsInThingGroup",
        "iot:CreateJob",
        "iot:DescribeJob",
        "iot:DescribeJobExecution",
        "iot:CancelJob"
      ],
      "Resource":[
        "arn:aws:iot:*:*:thing/idt-*",
        "arn:aws:iot:*:*:thinggroup/idt-*",
        "arn:aws:iot:*:*:policy/idt-*",
        "arn:aws:iot:*:*:cert/*",
        "arn:aws:iot:*:*:topic/idt-*",
        "arn:aws:iot:*:*:job/*"
      ]
    },
    {
      "Sid":"s3Resources",
      "Effect":"Allow",
```

```
      "Action":[
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObjectVersion",
        "s3:DeleteObject",
        "s3:CreateBucket",
        "s3:ListBucket",
        "s3:ListBucketVersions",
        "s3:DeleteBucket",
        "s3:PutObjectTagging",
        "s3:PutBucketTagging"
      ],
      "Resource":"arn:aws:s3::*:idt-*"
    },
    {
      "Sid":"roleAliasResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateRoleAlias",
        "iot:DescribeRoleAlias",
        "iot:DeleteRoleAlias",
        "iot:TagResource",
        "iam:GetRole"
      ],
      "Resource":[
        "arn:aws:iot:*:*:rolealias/idt-*",
        "arn:aws:iam::*:role/idt-*"
      ]
    },
    {
      "Sid":"idtExecuteAndCollectMetrics",
      "Effect":"Allow",
      "Action":[
        "iot-device-tester:SendMetrics",
        "iot-device-tester:SupportedVersion",
        "iot-device-tester:LatestIdt",
        "iot-device-tester:CheckVersion",
        "iot-device-tester:DownloadTestSuite"
      ],
      "Resource":"*"
    },
    {
      "Sid":"genericResources",
      "Effect":"Allow",
```

```
      "Action":[
        "greengrass:*",
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:ListThings",
        "iot:DescribeEndpoint",
        "iot:CreateKeysAndCertificate"
      ],
      "Resource":"*"
    },
    {
      "Sid":"iamResourcesUpdate",
      "Effect":"Allow",
      "Action":[
        "iam:CreateRole",
        "iam:DeleteRole",
        "iam:CreatePolicy",
        "iam:DeletePolicy",
        "iam:AttachRolePolicy",
        "iam:DetachRolePolicy",
        "iam:TagRole",
        "iam:TagPolicy",
        "iam:GetPolicy",
        "iam:ListAttachedRolePolicies",
        "iam:ListEntitiesForPolicy"
      ],
      "Resource":[
        "arn:aws:iam::*:role/idt-*",
        "arn:aws:iam::*:policy/idt-*"
      ]
    }
  ]
}
```

c. を使用している場合は PreInstalled、JSON タブでプレースホルダーコンテンツを次のポリシーに置き換えます。必ず次のことを行ってください。

- `iotResources` ステートメント内の *`thingName`* と *`thingGroup`* を、テスト対象デバイス (DUT) への Greengrass のインストール中に作成されたモノの名前とモノのグループに置き換えて、許可を追加します。

- roleAliasResources ステートメントおよび passRoleForResources ステートメントの *passRole* と *roleAlias* を、DUT への Greengrass のインストール中に作成されたロールに置き換えます。

```
{
    "Version":"2012-10-17",
    "Statement":[
    {
      "Sid":"passRoleForResources",
      "Effect":"Allow",
      "Action":"iam:PassRole",
      "Resource":"arn:aws:iam::*:role/passRole",
      "Condition":{
        "StringEquals":{
          "iam:PassedToService":[
            "iot.amazonaws.com",
            "lambda.amazonaws.com",
            "greengrass.amazonaws.com"
          ]
        }
      }
    },
    {
      "Sid":"lambdaResources",
      "Effect":"Allow",
      "Action":[
        "lambda:CreateFunction",
        "lambda:PublishVersion",
        "lambda:DeleteFunction",
        "lambda:GetFunction"
      ],
      "Resource":[
        "arn:aws:lambda:*:*:function:idt-*"
      ]
    },
    {
      "Sid":"iotResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateThing",
        "iot:DeleteThing",
        "iot:DescribeThing",
```

```
        "iot:CreateThingGroup",
        "iot:DeleteThingGroup",
        "iot:DescribeThingGroup",
        "iot:AddThingToThingGroup",
        "iot:RemoveThingFromThingGroup",
        "iot:AttachThingPrincipal",
        "iot:DetachThingPrincipal",
        "iot:UpdateCertificate",
        "iot:DeleteCertificate",
        "iot:CreatePolicy",
        "iot:AttachPolicy",
        "iot:DetachPolicy",
        "iot:DeletePolicy",
        "iot:GetPolicy",
        "iot:Publish",
        "iot:TagResource",
        "iot:ListThingPrincipals",
        "iot:ListAttachedPolicies",
        "iot:ListTargetsForPolicy",
        "iot:ListThingGroupsForThing",
        "iot:ListThingsInThingGroup",
        "iot:CreateJob",
        "iot:DescribeJob",
        "iot:DescribeJobExecution",
        "iot:CancelJob"
      ],
      "Resource":[
        "arn:aws:iot:*:*:thing/thingName",
        "arn:aws:iot:*:*:thinggroup/thingGroup",
        "arn:aws:iot:*:*:policy/idt-*",
        "arn:aws:iot:*:*:cert/*",
        "arn:aws:iot:*:*:topic/idt-*",
        "arn:aws:iot:*:*:job/*"
      ]
    },
    {
      "Sid":"s3Resources",
      "Effect":"Allow",
      "Action":[
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObjectVersion",
        "s3:DeleteObject",
        "s3:CreateBucket",
```

```
        "s3:ListBucket",
        "s3:ListBucketVersions",
        "s3:DeleteBucket",
        "s3:PutObjectTagging",
        "s3:PutBucketTagging"
      ],
      "Resource":"arn:aws:s3::*:idt-*"
    },
    {
      "Sid":"roleAliasResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateRoleAlias",
        "iot:DescribeRoleAlias",
        "iot:DeleteRoleAlias",
        "iot:TagResource",
        "iam:GetRole"
      ],
      "Resource":[
        "arn:aws:iot:*:*:rolealias/roleAlias",
        "arn:aws:iam::*:role/idt-*"
      ]
    },
    {
      "Sid":"idtExecuteAndCollectMetrics",
      "Effect":"Allow",
      "Action":[
        "iot-device-tester:SendMetrics",
        "iot-device-tester:SupportedVersion",
        "iot-device-tester:LatestIdt",
        "iot-device-tester:CheckVersion",
        "iot-device-tester:DownloadTestSuite"
      ],
      "Resource":"*"
    },
    {
      "Sid":"genericResources",
      "Effect":"Allow",
      "Action":[
        "greengrass:*",
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:ListThings",
        "iot:DescribeEndpoint",
```

```
        "iot:CreateKeysAndCertificate"
      ],
      "Resource":"*"
    },
    {
      "Sid":"iamResourcesUpdate",
      "Effect":"Allow",
      "Action":[
        "iam:CreateRole",
        "iam:DeleteRole",
        "iam:CreatePolicy",
        "iam:DeletePolicy",
        "iam:AttachRolePolicy",
        "iam:DetachRolePolicy",
        "iam:TagRole",
        "iam:TagPolicy",
        "iam:GetPolicy",
        "iam:ListAttachedRolePolicies",
        "iam:ListEntitiesForPolicy"
      ],
      "Resource":[
        "arn:aws:iam::*:role/idt-*",
        "arn:aws:iam::*:policy/idt-*"
      ]
    }
  ]
}
```

> **Note**
>
> テスト対象のデバイスのトークン交換ロールとしてカスタム IAM ロールを使用する場合は、カスタム IAM ロールリソースを許可するように、ポリシーの `roleAliasResources` ステートメントと `passRoleForResources` ステートメントを必ず更新してください。

d. [Review policy] (ポリシーの確認) を選択します。

e. [Name] (名前) に **`IDTGreengrassIAMPermissions`** と入力します。[概要] で、ポリシーによって付与されたアクセス許可を確認します。

f. [ポリシーの作成] を選択します。

3. IAM ユーザーを作成し、IDT for AWS IoT Greengrass に必要なアクセス許可をアタッチします。

   a. IAM ユーザーを作成します。IAM ユーザーガイドの IAM ユーザーの作成 (コンソール) のステップ 1 ~ 5 に従います。

   b. アクセス許可を IAM ユーザーにアタッチします。

      i. [Set permissions] ページで、[Attach existing policies to user directly] を選択します。

      ii. 前のステップで作成した IDTGreengrassIAMPermissions ポリシーを検索します。チェックボックスをオンにします。

   c. [Next: Tags] (次へ: タグ) を選択します。

   d. [Next: Review] (次へ: レビュー) を選択して、選択内容の概要を表示します。

   e. [ユーザーを作成] を選択します。

   f. ユーザーのアクセスキー (アクセスキー ID とシークレットアクセスキー) を表示するには、パスワードとアクセスキーの横にある [Show (表示)] を選択します。アクセスキーを保存するには、[Download .csv] を選択し、安全な場所にファイルを保存します。この情報を後で使用して、AWS 認証情報ファイルを設定します。

4. 次のステップ: 物理デバイスを設定します。

IDT 用のアクセス許可を設定するには (AWS CLI)

1. コンピュータに AWS CLI がまだインストールされていない場合は、インストールして設定します。AWS Command Line Interface ユーザーガイドの AWS CLI のインストールのステップに従います。

   > **Note**
   >
   > AWS CLI は、コマンドラインシェルから AWS のサービスとやり取りするために使用できるオープンソースツールです。

2. IDT と AWS IoT Greengrass ロールを管理するためのアクセス許可を付与するカスタマー管理ポリシーを作成します。

   a. を使用していない場合は PreInstalled、テキストエディタを開き、次のポリシー内容を JSON ファイルに保存します。を使用している場合は PreInstalled、次のステップに進みます。

```
{
    "Version":"2012-10-17",
    "Statement":[
    {
      "Sid":"passRoleForResources",
      "Effect":"Allow",
      "Action":"iam:PassRole",
      "Resource":"arn:aws:iam::*:role/idt-*",
      "Condition":{
        "StringEquals":{
          "iam:PassedToService":[
            "iot.amazonaws.com",
            "lambda.amazonaws.com",
            "greengrass.amazonaws.com"
          ]
        }
      }
    },
    {
      "Sid":"lambdaResources",
      "Effect":"Allow",
      "Action":[
        "lambda:CreateFunction",
        "lambda:PublishVersion",
        "lambda:DeleteFunction",
        "lambda:GetFunction"
      ],
      "Resource":[
        "arn:aws:lambda:*:*:function:idt-*"
      ]
    },
    {
      "Sid":"iotResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateThing",
        "iot:DeleteThing",
        "iot:DescribeThing",
        "iot:CreateThingGroup",
        "iot:DeleteThingGroup",
        "iot:DescribeThingGroup",
        "iot:AddThingToThingGroup",
        "iot:RemoveThingFromThingGroup",
```

```
        "iot:AttachThingPrincipal",
        "iot:DetachThingPrincipal",
        "iot:UpdateCertificate",
        "iot:DeleteCertificate",
        "iot:CreatePolicy",
        "iot:AttachPolicy",
        "iot:DetachPolicy",
        "iot:DeletePolicy",
        "iot:GetPolicy",
        "iot:Publish",
        "iot:TagResource",
        "iot:ListThingPrincipals",
        "iot:ListAttachedPolicies",
        "iot:ListTargetsForPolicy",
        "iot:ListThingGroupsForThing",
        "iot:ListThingsInThingGroup",
        "iot:CreateJob",
        "iot:DescribeJob",
        "iot:DescribeJobExecution",
        "iot:CancelJob"
      ],
      "Resource":[
        "arn:aws:iot:*:*:thing/idt-*",
        "arn:aws:iot:*:*:thinggroup/idt-*",
        "arn:aws:iot:*:*:policy/idt-*",
        "arn:aws:iot:*:*:cert/*",
        "arn:aws:iot:*:*:topic/idt-*",
        "arn:aws:iot:*:*:job/*"
      ]
    },
    {
      "Sid":"s3Resources",
      "Effect":"Allow",
      "Action":[
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObjectVersion",
        "s3:DeleteObject",
        "s3:CreateBucket",
        "s3:ListBucket",
        "s3:ListBucketVersions",
        "s3:DeleteBucket",
        "s3:PutObjectTagging",
        "s3:PutBucketTagging"
```

```
      ],
      "Resource":"arn:aws:s3::*:idt-*"
    },
    {
      "Sid":"roleAliasResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateRoleAlias",
        "iot:DescribeRoleAlias",
        "iot:DeleteRoleAlias",
        "iot:TagResource",
        "iam:GetRole"
      ],
      "Resource":[
        "arn:aws:iot:*:*:rolealias/idt-*",
        "arn:aws:iam::*:role/idt-*"
      ]
    },
    {
      "Sid":"idtExecuteAndCollectMetrics",
      "Effect":"Allow",
      "Action":[
        "iot-device-tester:SendMetrics",
        "iot-device-tester:SupportedVersion",
        "iot-device-tester:LatestIdt",
        "iot-device-tester:CheckVersion",
        "iot-device-tester:DownloadTestSuite"
      ],
      "Resource":"*"
    },
    {
      "Sid":"genericResources",
      "Effect":"Allow",
      "Action":[
        "greengrass:*",
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:ListThings",
        "iot:DescribeEndpoint",
        "iot:CreateKeysAndCertificate"
      ],
      "Resource":"*"
    },
    {
```

```
      "Sid":"iamResourcesUpdate",
      "Effect":"Allow",
      "Action":[
        "iam:CreateRole",
        "iam:DeleteRole",
        "iam:CreatePolicy",
        "iam:DeletePolicy",
        "iam:AttachRolePolicy",
        "iam:DetachRolePolicy",
        "iam:TagRole",
        "iam:TagPolicy",
        "iam:GetPolicy",
        "iam:ListAttachedRolePolicies",
        "iam:ListEntitiesForPolicy"
      ],
      "Resource":[
        "arn:aws:iam::*:role/idt-*",
        "arn:aws:iam::*:policy/idt-*"
      ]
    }
  ]
}
```

b. を使用している場合は PreInstalled、テキストエディタを開き、次のポリシー内容を JSON ファイルに保存します。必ず次のことを行ってください。

- テスト対象デバイス (DUT) への Greengrass のインストール中に作成された `iotResources` ステートメント内の *`thingName`* と *`thingGroup`* を置き換えて、許可を追加します。
- `roleAliasResources` ステートメントおよび `passRoleForResources` ステートメントの *`passRole`* と *`roleAlias`* を、DUT への Greengrass のインストール中に作成されたロールに置き換えます。

```
{
    "Version":"2012-10-17",
    "Statement":[
    {
      "Sid":"passRoleForResources",
      "Effect":"Allow",
      "Action":"iam:PassRole",
      "Resource":"arn:aws:iam::*:role/passRole",
```

```
      "Condition":{
        "StringEquals":{
          "iam:PassedToService":[
            "iot.amazonaws.com",
            "lambda.amazonaws.com",
            "greengrass.amazonaws.com"
          ]
        }
      }
    },
    {
      "Sid":"lambdaResources",
      "Effect":"Allow",
      "Action":[
        "lambda:CreateFunction",
        "lambda:PublishVersion",
        "lambda:DeleteFunction",
        "lambda:GetFunction"
      ],
      "Resource":[
        "arn:aws:lambda:*:*:function:idt-*"
      ]
    },
    {
      "Sid":"iotResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateThing",
        "iot:DeleteThing",
        "iot:DescribeThing",
        "iot:CreateThingGroup",
        "iot:DeleteThingGroup",
        "iot:DescribeThingGroup",
        "iot:AddThingToThingGroup",
        "iot:RemoveThingFromThingGroup",
        "iot:AttachThingPrincipal",
        "iot:DetachThingPrincipal",
        "iot:UpdateCertificate",
        "iot:DeleteCertificate",
        "iot:CreatePolicy",
        "iot:AttachPolicy",
        "iot:DetachPolicy",
        "iot:DeletePolicy",
        "iot:GetPolicy",
```

```
        "iot:Publish",
        "iot:TagResource",
        "iot:ListThingPrincipals",
        "iot:ListAttachedPolicies",
        "iot:ListTargetsForPolicy",
        "iot:ListThingGroupsForThing",
        "iot:ListThingsInThingGroup",
        "iot:CreateJob",
        "iot:DescribeJob",
        "iot:DescribeJobExecution",
        "iot:CancelJob"
      ],
      "Resource":[
        "arn:aws:iot:*:*:thing/thingName",
        "arn:aws:iot:*:*:thinggroup/thingGroup",
        "arn:aws:iot:*:*:policy/idt-*",
        "arn:aws:iot:*:*:cert/*",
        "arn:aws:iot:*:*:topic/idt-*",
        "arn:aws:iot:*:*:job/*"
      ]
    },
    {
      "Sid":"s3Resources",
      "Effect":"Allow",
      "Action":[
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObjectVersion",
        "s3:DeleteObject",
        "s3:CreateBucket",
        "s3:ListBucket",
        "s3:ListBucketVersions",
        "s3:DeleteBucket",
        "s3:PutObjectTagging",
        "s3:PutBucketTagging"
      ],
      "Resource":"arn:aws:s3::*:idt-*"
    },
    {
      "Sid":"roleAliasResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateRoleAlias",
        "iot:DescribeRoleAlias",
```

```
        "iot:DeleteRoleAlias",
        "iot:TagResource",
        "iam:GetRole"
      ],
      "Resource":[
        "arn:aws:iot:*:*:rolealias/roleAlias",
        "arn:aws:iam::*:role/idt-*"
      ]
    },
    {
      "Sid":"idtExecuteAndCollectMetrics",
      "Effect":"Allow",
      "Action":[
        "iot-device-tester:SendMetrics",
        "iot-device-tester:SupportedVersion",
        "iot-device-tester:LatestIdt",
        "iot-device-tester:CheckVersion",
        "iot-device-tester:DownloadTestSuite"
      ],
      "Resource":"*"
    },
    {
      "Sid":"genericResources",
      "Effect":"Allow",
      "Action":[
        "greengrass:*",
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:ListThings",
        "iot:DescribeEndpoint",
        "iot:CreateKeysAndCertificate"
      ],
      "Resource":"*"
    },
    {
      "Sid":"iamResourcesUpdate",
      "Effect":"Allow",
      "Action":[
        "iam:CreateRole",
        "iam:DeleteRole",
        "iam:CreatePolicy",
        "iam:DeletePolicy",
        "iam:AttachRolePolicy",
        "iam:DetachRolePolicy",
```

```
        "iam:TagRole",
        "iam:TagPolicy",
        "iam:GetPolicy",
        "iam:ListAttachedRolePolicies",
        "iam:ListEntitiesForPolicy"
      ],
      "Resource":[
        "arn:aws:iam::*:role/idt-*",
        "arn:aws:iam::*:policy/idt-*"
      ]
    }
  ]
}
```

> **Note**
>
> テスト対象のデバイスのトークン交換ロールとしてカスタム IAM ロールを使用する場合は、カスタム IAM ロールリソースを許可するように、ポリシーの `roleAliasResources` ステートメントと `passRoleForResources` ステートメントを必ず更新してください。

c. 次のコマンドを実行して、`IDTGreengrassIAMPermissions` という名前でカスタマー管理ポリシーを作成します。*`policy.json`* を前のステップで作成した JSON ファイルへのフルパスに置き換えます。

```
aws iam create-policy --policy-name IDTGreengrassIAMPermissions --policy-
document file://policy.json
```

3. IAM ユーザーを作成し、IDT for AWS IoT Greengrass に必要なアクセス許可をアタッチします。

a. IAM ユーザーを作成します。このセットアップ例では、ユーザーは `IDTGreengrassUser` という名前になります。

```
aws iam create-user --user-name IDTGreengrassUser
```

b. ステップ 2 で作成した `IDTGreengrassIAMPermissions` ポリシーを IAM ユーザーにアタッチします。コマンドの *`<account-id>`* を AWS アカウント の ID に置き換えます。

```
aws iam attach-user-policy --user-name IDTGreengrassUser --policy-arn
 arn:aws:iam::<account-id>:policy/IDTGreengrassIAMPermissions
```

4. ユーザーのシークレットアクセスキーを作成します。

   ```
   aws iam create-access-key --user-name IDTGreengrassUser
   ```

   この出力は安全な場所に保存してください。この情報を後で使用して、AWS 認証情報ファイルを設定します。

5. 次のステップ: 物理デバイスを設定します。

AWS IoT Device Tester のアクセス許可

次のポリシーは、AWS IoT Device Tester の権限を表します。

AWS IoT Device Tester のバージョンチェックと自動更新機能には、これらの権限が必要です。

- `iot-device-tester:SupportedVersion`

  対応する製品、テストスイート、IDT バージョン一覧を取得する AWS IoT Device Tester 権限を付与します。

- `iot-device-tester:LatestIdt`

  ダウンロード可能な最新の IDT バージョンを取得する AWS IoT Device Tester 権限を付与します。

- `iot-device-tester:CheckVersion`

  IDT、テストスイート、および製品のバージョン互換性を確認する AWS IoT Device Tester 権限を付与します。

- `iot-device-tester:DownloadTestSuite`

  テストスイートのアップデートをダウンロードする AWS IoT Device Tester 権限を付与します。

また、AWS IoT Device Tester はオプションのメトリックレポートのために、次の権限を使用します。

- `iot-device-tester:SendMetrics`

AWS IoT Device Tester 内部の使用状況に関するメトリックを収集する権限を AWS に付与します。この権限を省略すると、これらのメトリックは収集されません。

IDT テストを実行するようにデバイスを設定する

IDT でデバイス認定のためのテストを実行するには、デバイスにアクセスするようにホストコンピュータを設定して、デバイス上でユーザーのアクセス許可を設定する必要があります。

ホストコンピュータに Java をインストールする

IDT v4.2.0 以降、オプションの AWS IoT Greengrass に対する認定テストでは Java を実行する必要があります。

Java バージョン 8 以降を使用することができます。Amazon Corretto または OpenJDK の長期サポートバージョンを使用することをお勧めします。バージョン 8 以降が必要です。

テスト対象デバイスにアクセスするようにホストコンピュータを設定する

IDT はホストコンピュータで動作し、SSH を使用してデバイスに接続できる必要があります。IDT がテスト対象のデバイスへの SSH アクセスを許可するには、2 つのオプションがあります。

1. こちらの手順に従って SSH キーペアを作成し、パスワードを指定せずにテスト対象のデバイスにサインインすることをキーに承認します。
2. `device.json` ファイルに各デバイスのユーザー名とパスワードを入力します。詳細については、「device.json の設定」を参照してください。

任意の SSL 実装を使用して SSH キーを作成できます。次の手順は、SSH-KEYGEN または PuTTYgen (Windows の場合) を使用する方法を示しています。別の SSL 実装を使用する場合は、その実装に関するドキュメントを参照してください。

テスト対象デバイスで認証するには、IDT で SSH キーを使用します。

SSH-KEYGEN を使用して SSH キーを作成するには

1. SSH キーを作成します。

   OpenSSH ssh-keygen コマンドを使用して SSH キーペアを作成できます。ホストコンピュータに SSH キーペアがすでにある場合は、IDT 専用の SSH キーペアを作成することをお勧めしま

す。こうすることで、テストを完了した後、ホストコンピュータはパスワードを入力しないとデバイスに接続できなくなります。また、リモートデバイスへのアクセスを必要なユーザーのみに制限することもできます。

> **Note**
>
> Windows に SSH クライアントがインストールされていません。Windows での SSH クライアントのインストールについては、「SSH クライアントソフトウェアをダウンロードする」を参照してください。

ssh-keygen コマンドは、キーペアを保存する名前とパスの入力を求めます。デフォルトでは、キーペアファイルの名前は `id_rsa` (プライベートキー) と `id_rsa.pub` (パブリックキー) です。macOS および Linux の場合、これらのファイルのデフォルトの場所は `~/.ssh/` です。Windows の場合、デフォルトの場所は `C:\Users\<user-name>\.ssh` です。

プロンプトが表示されたら、SSH キーを保護するキーフレーズを入力します。詳細については、「新しい SSH キーを生成する」を参照してください。

2. テスト対象デバイスに承認済み SSH キーを追加します。

   IDT で SSH プライベートキーを使用して、テスト対象デバイスにサインインする必要があります。SSH プライベートキーがテスト対象デバイスにサインインすることを承認するには、ホストコンピュータから ssh-copy-id コマンドを使用します。このコマンドは、テスト対象デバイスの `~/.ssh/authorized_keys` ファイルにパブリックキーを追加します。例:

   $ ssh-copy-id ***<remote-ssh-user>***@***<remote-device-ip>***

   ここで、 *remote-ssh-user* はテスト対象のデバイスにサインインするために使用されるユーザー名であり、 *remote-device-ip*はテスト対象のデバイスの IP アドレスであり、テストを実行します。例:

   ssh-copy-id pi@192.168.1.5

   プロンプトが表示されたら、ssh-copy-id コマンドで指定したユーザー名に対応するパスワードを入力します。

   ssh-copy-id では、パブリックキー名が `id_rsa.pub` で、デフォルトの保存先が `~/.ssh/` (macOS と Linux の場合) または `C:\Users\<user-name>\.ssh` (Windows の場合) であるとみなされます。パブリックキーに別の名前や別の保存先を指定した場合は、ssh-copy-id で -

i オプションを使用し、SSH 公開鍵への完全修飾パス (ssh-copy-id -i ~/my/path/myKey.pub など) を指定する必要があります。SSH キーの作成とパブリックキーのコピーの詳細については、「SSH-COPY-ID」を参照してください。

PuTTYgen を使用して SSH キーを作成するには (Windows のみ)

1. テスト対象デバイスに OpenSSH サーバーとクライアントがインストールされていることを確認します。詳細については、「OpenSSH」を参照してください。
2. テスト対象のデバイスに PuTTYgen をインストールします。
3. PuTTYGen を開きます。
4. [Generate] を選択し、ボックス内にマウスカーソルを移動してプライベートキーを生成します。
5. [Conversions] メニューから [Export OpenSSH key] を選択し、プライベートキーに `.pem` ファイル拡張子を付けて保存します。
6. テスト対象デバイスの `/home/<user>/.ssh/authorized_keys` ファイルにパブリックキーを追加します。

   a. PuTTYgen ウィンドウからパブリックキーテキストをコピーします。

   b. PuTTY を使用して、テスト対象のデバイスでセッションを作成します。

      i. コマンドプロンプトまたは Windows Powershell ウィンドウから、次のコマンドを実行します。

         C:/***<path-to-putty>***/putty.exe -ssh ***<user>***@***<dut-ip-address>***

      ii. プロンプトが表示されたら、デバイスのパスワードを入力します。

      iii. vi などのテキストエディタを使用して、テスト対象のデバイスの `/home/<user>/.ssh/authorized_keys` ファイルにパブリックキーを追加します。
7. `device.json` ファイルを、ユーザー名、IP アドレス、およびテスト対象の各デバイスのホストコンピュータに保存したプライベートキーファイルへのパスで更新します。詳細については、「the section called “device.json の設定”」を参照してください。必ずプライベートキーの完全パスとファイル名を指定し、スラッシュ (「/」) を使用してください。たとえば、Windows パス `C:\DT\privatekey.pem` の場合は、`device.json` ファイルで `C:/DT/privatekey.pem` を使用します。

Windows デバイスのユーザー認証情報を設定する

Windows ベースのデバイスを認定するには、テスト対象のデバイスの LocalSystem アカウントで、次のユーザーのユーザー認証情報を設定する必要があります。

- デフォルトの Greengrass ユーザー (ggc_user)。
- テスト対象のデバイスに接続するために使用するユーザー。このユーザは、device.json ファイルで設定します。

テスト対象のデバイスの LocalSystem アカウントで各ユーザーを作成し、そのユーザーのユーザー名とパスワードをLocalSystem アカウントの認証情報マネージャーインスタンスに保存する必要があります。

Windows デバイスでユーザーを設定するには

1. 管理者として Windows コマンドプロンプト cmd.exe を開きます。
2. Windows デバイスの LocalSystem アカウントにユーザーを作成します。作成する各ユーザーに対して、次のコマンドを実行します。デフォルトの Greengrass ユーザーの場合は、*user-name* を ggc_user と置き換えます。*#####*を安全なパスワードに置き換えます。

   ```
   net user /add user-name password
   ```

3. デバイスに Microsoft からPsExecユーティリティをダウンロードしてインストールします。
4. PsExec ユーティリティを使用して、デフォルトユーザーのユーザー名とパスワードを LocalSystem アカウントの認証情報マネージャーインスタンスに保存します。

   認証情報マネージャーで、設定する各ユーザーに対して、次のコマンドを実行します。デフォルトの Greengrass ユーザーの場合は、*user-name* を ggc_user と置き換えます。*#####*を以前に設定したユーザーのパスワードに置き換えます。

   ```
   psexec -s cmd /c cmdkey /generic:user-name /user:user-name /pass:password
   ```

   PsExec License Agreement が開いたら、Accept を選択し、ライセンスに同意してコマンドを実行します。

> **Note**
>
> Windows デバイスでは、 LocalSystem アカウントは Greengrass nucleus を実行するため、 PsExec ユーティリティを使用してユーザー情報を LocalSystemアカウントに格納する必要があります。認証情報マネージャーアプリケーションを使用すると、この情報は アカウントではなく、現在ログオンしているユーザーの Windows LocalSystemアカウントに保存されます。

デバイスに対するユーザーのアクセス許可を設定する

IDT は、テスト対象デバイスのさまざまなディレクトリやファイルに対してオペレーションを実行します。このようなオペレーションの中には、高いアクセス許可が必要な場合があります (sudo を使用)。これらの操作を自動化するには、IDT for AWS IoT Greengrass V2 がパスワードの入力を求められることなく、sudo を使用してコマンドを実行できる必要があります。

パスワードの入力を求めることなく、sudo にアクセスを許可するには、テスト対象デバイスで以下の手順を実行します。

> **Note**
>
> `username` は、テスト対象デバイスにアクセスするために IDT で使用する SSH ユーザーを指します。

ユーザーを sudo グループに追加するには

1. テスト対象のデバイスで、`sudo usermod -aG sudo` *`<username>`* を実行します。
2. サインアウトし、再度サインインして、変更を反映します。
3. ユーザー名が正常に追加されたことを確認するには、sudo echo test を実行します。パスワードの入力を要求されない場合、ユーザーは正しく設定されています。
4. `/etc/sudoers` ファイルを開き、ファイルの末尾に次の行を追加します:

   ```
   <ssh-username> ALL=(ALL) NOPASSWD: ALL
   ```

カスタムトークン交換ロールを設定する

テスト対象のデバイスが AWS リソースとやり取りを継承するトークン交換ロールとして、カスタム IAM ロールの使用を選択することができます。IAM ロールの作成方法の詳細については、「IAM ユーザーガイド」の「IAM ロールを作成する」を参照してください。

IDT がカスタム IAM ロールを使用できるようにするには、以下の要件を満たす必要があります。このロールには、最低限必要なポリシーアクションのみを追加することを強く推奨します。

- userdata.json 設定ファイルを更新して、`GreengrassV2TokenExchangeRole` パラメータを `true` に設定する必要があります。
- カスタム IAM ロールは、次の最小限の信頼ポリシーで設定する必要があります。

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Principal":{
            "Service":[
               "credentials.iot.amazonaws.com",
               "lambda.amazonaws.com",
               "sagemaker.amazonaws.com"
            ]
         },
         "Action":"sts:AssumeRole"
      }
   ]
}
```

- カスタム IAM ロールは、次の最小限のアクセス許可ポリシーで設定する必要があります。

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "iot:DescribeCertificate",
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents",
```

```
            "logs:DescribeLogStreams",
            "iot:Connect",
            "iot:Publish",
            "iot:Subscribe",
            "iot:Receive",
            "iot:ListThingPrincipals",
            "iot:GetThingShadow",
            "iot:UpdateThingShadow",
            "s3:GetBucketLocation",
            "s3:GetObject",
            "s3:PutObject",
            "s3:AbortMultipartUpload",
            "s3:ListMultipartUploadParts"
         ],
         "Resource":"*"
      }
   ]
}
```

- カスタム IAM ロールの名前は、テストユーザーの IAM アクセス許可で指定した IAM ロールリソースと一致する必要があります。デフォルトでは、テストユーザーポリシーは、ロール名に `idt-` プレフィックスを持つ IAM ロールへのアクセスを許可します。IAM ロール名にこのプレフィックスが使用されていない場合、次の例に示されているように、テストユーザーポリシーの `roleAliasResources` ステートメントと `passRoleForResources` ステートメントに `arn:aws:iam::*:role/custom-iam-role-name` リソースを追加してください。

  Example **`passRoleForResources`** ステートメント

```
{
   "Sid":"passRoleForResources",
   "Effect":"Allow",
   "Action":"iam:PassRole",
   "Resource":"arn:aws:iam::*:role/custom-iam-role-name",
   "Condition":{
      "StringEquals":{
         "iam:PassedToService":[
            "iot.amazonaws.com",
            "lambda.amazonaws.com",
            "greengrass.amazonaws.com"
         ]
      }
   }
```

```
}
```

Example **`roleAliasResources`** ステートメント

```
{
   "Sid":"roleAliasResources",
   "Effect":"Allow",
   "Action":[
      "iot:CreateRoleAlias",
      "iot:DescribeRoleAlias",
      "iot:DeleteRoleAlias",
      "iot:TagResource",
      "iam:GetRole"
   ],
   "Resource":[
      "arn:aws:iot:*:*:rolealias/idt-*",
      "arn:aws:iam::*:role/custom-iam-role-name"
   ]
}
```

オプション機能をテストするためのデバイスの設定

このセクションでは、オプションの Docker および機械学習 (ML) 機能に対して IDT テストを実行する際のデバイス要件について説明します。これらの機能をテストする場合にのみ、デバイスがこれらの要件を満たしていることを確認する必要があります。それ以外の場合は、「the section called “IDT 設定を設定する”」に進みます。

トピック

- Docker の認定要件
- ML の認定要件
- HSM の認定要件

Docker の認定要件

IDT for AWS IoT Greengrass V2 は、カスタム Docker コンテナコンポーネントを使って実行できる Docker コンテナイメージをダウンロードすることが可能な、AWS から提供される Docker アプリケーションマネージャーコンポーネントを、デバイスが使用できることを検証するための Docker 認

定テストを提供しています。カスタム Docker コンポーネントを作成するための詳細については、「Docker コンテナの実行」を参照してください。

Docker 認定テストを実行するには、テスト対象のデバイスが Docker アプリケーションマネージャーコンポーネントをデプロイするための次の要件を満たしている必要があります。

- Docker Engine 1.9.1 以降が Greengrass コアにインストールされていいること。バージョン 20.10 は、AWS IoT Greengrass Core ソフトウェアとの動作が確認されている最新バージョンです。Docker コンテナを実行するコンポーネントをデプロイする前に、コアデバイスに直接、Docker をインストールしておく必要があります。
- このコンポーネントをデプロイする前に、Docker デーモンがコアデバイス上で起動し、実行されています。
- Docker コンテナコンポーネントを実行するシステムユーザーには、ルート権限または管理者権限が必要です。権限がない場合は、ルート権限または管理者権限を持たないユーザーとして実行されるように Docker を設定する必要があります。
  - Linux デバイスでは、ユーザーを `docker` グループに追加することで、`sudo` のない `docker` コマンドを呼び出せます。
  - Windows デバイスでは、ユーザーを `docker-users` グループ に追加することで、管理者の権限のない `docker` コマンドを呼び出せます。

  Linux or Unix

  Docker コンテナコンポーネントの実行に使用する `ggc_user` または非ルートユーザーを `docker` グループに追加するには、次のコマンドを実行します。

  ```
  sudo usermod -aG docker ggc_user
  ```

  詳細については、「Docker を非ルートユーザーとして管理する」を参照してください。

  Windows Command Prompt (CMD)

  Docker コンテナコンポーネントの実行に使用する `ggc_user` またはユーザーを `docker-users` グループに追加するには、次のコマンドを管理者として実行します。

  ```
  net localgroup docker-users ggc_user /add
  ```

Windows PowerShell

Docker コンテナコンポーネントの実行に使用する ggc_user またはユーザーを docker-users グループに追加するには、次のコマンドを管理者として実行します。

```
Add-LocalGroupMember -Group docker-users -Member ggc_user
```

ML の認定要件

IDT for AWS IoT Greengrass V2 には、 AWSが提供する機械学習コンポーネントを使用して、デバイスが Deep Learning Runtime または TensorFlow Lite ML フレームワークを使用してローカルで ML 推論を実行できることを検証するための ML 認定テストが用意されています。Greengrass デバイスで ML 推論を実行する際の詳細については、「機械学習の推論を実行する」を参照してください。

ML 認定テストを実行するには、テスト対象のデバイスが機械学習コンポーネントをデプロイするための次の要件を満たしている必要があります。

- Amazon Linux 2 または Ubuntu 18.04 を実行している Greengrass コアデバイスの場合は、GNU C ライブラリ (glibc) バージョン 2.27 以降がデバイスにインストールされている必要があります。
- Raspberry Pi などの Armv7l デバイスでは、OpenCV-Python の依存関係がデバイスにインストールされています。次のコマンドを実行して依存関係をインストールします。

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

- Raspberry Pi OS Bullseye を実行する Raspberry Pi デバイスでは、次の要件を満たす必要があります。
  - NumPy 1.22.4 以降がデバイスにインストールされている。Raspberry Pi OS Bullseye には以前のバージョンの が含まれているため NumPy、次のコマンドを実行して NumPy デバイスでアップグレードできます。

    ```
    pip3 install --upgrade numpy
    ```

  - デバイスで、レガシーカメラスタックが有効になっていること。Raspberry Pi OS Bullseye には、デフォルトで新しいカメラスタックが含まれており有効化されていますが、これには互換性がないるため、レガシーカメラスタックを有効にしておく必要があります。

レガシーカメラスタックを有効にするには

1. 次のコマンドを実行して、Raspberry Pi 設定ツールを開きます。

```
sudo raspi-config
```

2. [Interface Options] (インターフェイスオプション) を選択します。
3. [Legacy camera] (レガシーカメラ) を選択して、レガシーカメラスタックを有効にします。
4. Raspberry Pi を再起動します。

HSM の認定要件

AWS IoT Greengrass は、デバイス上の PKCS ハードウェアセキュリティモジュール (HSM) と統合するために必要な PKCS#11 プロバイダコンポーネントを提供しています。HSM の設定は、お使いのデバイスと、選択した HSM モジュールによって異なります。IDT 構成設定に文書化されているように、予想される HSM 設定が提供される限り、IDT は、このオプション機能である認定テストを実行するために必要な情報を入手することができます。

AWS IoT Greengrass 認定スイートを実行するための IDT 設定を設定する

テストを実行する前に、ホストコンピュータの AWS 認証情報およびデバイスの設定を設定する必要があります。

config.json で AWS 認証情報を設定する

IAM ユーザー認証情報を *<device_tester_extract_location>*/configs/config.json ファイルで設定する必要があります。the section called “AWS アカウント を作成して設定する” で作成した IDT for AWS IoT Greengrass V2 ユーザーの認証情報を使用します。以下のいずれかの方法で認証情報を指定できます。

- 認証情報ファイルを使用する
- 環境変数を使用する

認証情報ファイルを使用して AWS 認証情報を設定する

IDT では、AWS CLI と同じ認証情報ファイルが使用されます。詳細については、「設定ファイルと認証情報ファイル」を参照してください。

認証情報ファイルの場所は、使用しているオペレーティングシステムによって異なります。

- macOS、Linux: `~/.aws/credentials`
- Windows: `C:\Users\UserName\.aws\credentials`

AWS 認証情報を次の形式で `credentials` ファイルに追加します。

```
[default]
aws_access_key_id = <your_access_key_id>
aws_secret_access_key = <your_secret_access_key>
```

`credentials` ファイルの AWS 認証情報を使用するように IDT for AWS IoT Greengrass V2 を設定するには、`config.json` ファイルを次のように編集します。

```
{
  "awsRegion": "region",
  "auth": {
    "method": "file",
    "credentials": {
      "profile": "default"
    }
  }
}
```

> **Note**
>
> `default` の AWS ファイルを使用しない場合は、必ず `config.json` ファイルのプロファイル名を変更してください。詳細については、「名前付きプロファイル」を参照してください。

環境変数を使用して AWS 認証情報を設定する

環境変数は、オペレーティングシステムによって維持され、システムコマンドによって使用される変数です。SSH セッションを閉じると、これらは保存されません。IDT for AWS IoT Greengrass V2 は、`AWS_ACCESS_KEY_ID` と `AWS_SECRET_ACCESS_KEY` という環境変数を使用して AWS 認証情報を保存します。

これらの変数を Linux、macOS、または Unix で設定するには、export を使用します。

```
export AWS_ACCESS_KEY_ID=<your_access_key_id>
export AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

Windows でこれらの変数を設定するには、set を使用します。

```
set AWS_ACCESS_KEY_ID=<your_access_key_id>
set AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

環境変数を使用するように IDT を設定するには、config.json ファイルの auth セクションを編集します。以下がその例です。

```
{
  "awsRegion": "region",
  "auth": {
    "method": "environment"
  }
}
```

device.json の設定

AWS 認証情報に加えて、AWS IoT Greengrass V2 の IDTは、テストが実行されるデバイスに関する情報を必要とします。情報の例としては、IP アドレス、ログイン情報、オペレーティングシステム、CPU アーキテクチャなどがあります。

これらの情報を指定するには、 *<device_tester_extract_location>*/configs/device.json にある device.json テンプレートを使用する必要があります。

```
[
  {
    "id": "<pool-id>",
    "sku": "<sku>",
    "features": [
      {
        "name": "arch",
        "value": "x86_64 | armv6l | armv7l | aarch64"
      },
      {
        "name": "ml",
        "value": "dlr | tensorflowlite | dlr,tensorflowlite | no"
      },
      {
```

```
        "name": "docker",
        "value": "yes | no"
      },
      {
        "name": "streamManagement",
        "value": "yes | no"
      },
      {
        "name": "hsi",
        "value": "hsm | no"
      }
    ],
    "devices": [
      {
        "id": "<device-id>",
        "operatingSystem": "Linux | Windows",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": 22,
          "publicKeyPath": "<public-key-path>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
]
```

> **Note**
>
> method が pki に設定されている場合のみ privKeyPath を指定します。
> method が password に設定されている場合のみ password を指定します。

以下に説明するように、値が含まれているすべてのプロパティは必須です。

`id`

デバイスプールと呼ばれるデバイスのコレクションを一意に識別するユーザー定義の英数字の ID。プールに属するデバイスには、同一のハードウェアが必要です。テストスイートを実行する場合、プールのデバイスを使用してワークロードを並列化します。複数のデバイスを使用して異なるテストを実行します。

`sku`

テスト対象デバイスを一意に識別する英数字の値。SKU は、適格性が確認されたボードの追跡に使用されます。

> **Note**
>
> AWS Partner Device Catalog にデバイスを出品する場合は、ここで指定する SKU と出品プロセスで使用する SKU が一致しなければなりません。

`features`

デバイスでサポートされている機能を含む配列。すべての機能が必要です。

`arch`

テスト実行で検証される、サポートされているオペレーティングシステムアーキテクチャ。有効な値は次のとおりです。

- `x86_64`
- `armv6l`
- `armv7l`
- `aarch64`

`ml`

デバイスが、AWS が提供する機械学習 (ML) コンポーネントを使用するために必要な、すべての技術的依存関係を満たしていることを検証します。

この機能を有効にすると、デバイスが Deep Learning Runtime と TensorFlow Lite ML フレームワークを使用して ML 推論を実行できることも検証されます。

有効な値は、`dlr` と `tensorflowlite`、または `no` の任意の組み合わせです。

`docker`

デバイスが、AWS が提供する Docker アプリケーションマネージャー (`aws.greengrass.DockerApplicationManager`) コンポーネントを使用するために必要な、すべての技術的依存関係を満たしていることを検証します。

この機能を有効にすると、デバイスが Amazon ECR から Docker コンテナイメージをダウンロードできることも検証されます。

有効な値は、`yes` または `no` の任意の組み合わせです。

`streamManagement`

デバイスが AWS IoT Greengrass ストリームマネージャーをダウンロード、インストール、および実行できることを検証します。

有効な値は、`yes` または `no` の任意の組み合わせです。

`hsi`

デバイスが、ハードウェアセキュリティモジュール (HSM) に保存されているプライベートキーおよび証明書を使用して、AWS IoT および AWS IoT Greengrass のサービスへの接続を認証できることを検証します。またこのテストは、AWS が提供する PKCS #11 プロバイダコンポーネントが、ベンダーが提供する PKCS #11 ライブラリを使用することで HSM と連携できることも検証します。詳細については、「ハードウェアセキュリティ統合」を参照してください。

有効な値は `hsm` または `no` です。

> **Note**
>
> IDT v4.2.0 以降のバージョンでは、`ml`、`docker`、および `streamManagement` の機能のテストをサポートしています。これらの機能をテストしない場合は、対応する値を `no` に設定します。

> **Note**
>
> `hsi` のテストは IDT v4.5.1 以降のバージョンでのみ利用できます。

`devices.id`

テスト対象のデバイスのユーザー定義の一意の識別子。

`devices.operatingSystem`

デバイスのオペレーティングシステム。サポートされている値は、`Linux` および `Windows` です。

`connectivity.protocol`

このデバイスと通信するために使用される通信プロトコル。現在、サポートされている値は物理デバイスで `ssh` のみです。

`connectivity.ip`

テスト対象のデバイスの IP アドレス。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.port`

オプションです。SSH 接続に使用するポート番号。

デフォルト値は 22 です。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.publicKeyPath`

オプションです。テスト対象のデバイスへの接続を認証するために使用される公開キーへのフルパス。

`publicKeyPath` を指定すると、IDT がテスト対象のデバイスへの SSH 接続を確立するときに、デバイスの公開キーを検証します。この値が指定されていない場合、IDT は SSH 接続を作成しますが、デバイスのパブリックキーは検証しません。

公開キーへのパスを指定し、安全な方法を使用して、この公開キーをフェッチすることを強くお勧めします。標準のコマンドラインベースの SSH クライアントの場合、パブリックキーは `known_hosts` ファイルで提供されます。別の公開キーファイルを指定する場合、このファイルは `known_hosts` ファイルと同じ形式、つまり (*`ip-address key-type public-key`*) を使用する必要があります。同じ ip-address を持つエントリが複数ある場合、IDT で使用されるキータイプのエントリは、ファイル内の他のエントリよりも前である必要があります。

`connectivity.auth`

接続の認証情報。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.auth.method`

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- `pki`
- `password`

`connectivity.auth.credentials`

認証に使用される認証情報。

`connectivity.auth.credentials.password`

テスト中のデバイスにサインインするためのパスワード。

この値は、`connectivity.auth.method` が `password` に設定されている場合にのみ適用されます。

`connectivity.auth.credentials.privKeyPath`

テスト中のデバイスにサインインするためのプライベートキーへの完全パス。

この値は、`connectivity.auth.method` が `pki` に設定されている場合にのみ適用されます。

`connectivity.auth.credentials.user`

テスト対象デバイスにサインインするためのユーザー名。

userdata.json を設定する

IDT for AWS IoT Greengrass V2 では、テストアーティファクトと AWS IoT Greengrass ソフトウェアの場所に関する追加情報も必要です。

これらの情報を指定するには、 *<device_tester_extract_location>*`/configs/userdata.json` にある `userdata.json` テンプレートを使用する必要があります。

```
{
    "TempResourcesDirOnDevice": "/path/to/temp/folder",
    "InstallationDirRootOnDevice": "/path/to/installation/folder",
    "GreengrassNucleusZip": "/path/to/aws.greengrass.nucleus.zip",
    "PreInstalled": "yes/no",
    "GreengrassV2TokenExchangeRole": "custom-iam-role-name",
 "hsm": {
        "greengrassPkcsPluginJar": "/path/to/aws.greengrass.crypto.Pkcs11Provider-
latest.jar",
        "pkcs11ProviderLibrary": "/path/to/pkcs11-vendor-library",
        "slotId": "slot-id",
        "slotLabel": "slot-label",
        "slotUserPin": "slot-pin",
        "keyLabel": "key-label",
        "preloadedCertificateArn": "certificate-arn"
        "rootCA": "path/to/root-ca"
    }
}
```

以下に説明するように、値が含まれているすべてのプロパティは必須です。

`TempResourcesDirOnDevice`

テストアーティファクトを保存する、テスト対象のデバイスの一時フォルダへのフルパス。このディレクトリへの書き込みに sudo 権限が必要ないことを確認してください。

> **Note**
>
> IDT は、テストの実行が終了すると、このフォルダのコンテンツを削除します。

`InstallationDirRootOnDevice`

AWS IoT Greengrass をインストールするデバイスのフォルダへのフルパス。PreInstalled Greengrass の場合、これは Greengrass インストールディレクトリへのパスです。

このフォルダに必要なファイル権限を設定する必要があります。インストールパス内のフォルダごとに、以下のコマンドを実行します。

```
sudo chmod 755 folder-name
```

`GreengrassNucleusZip`

ホストコンピュータの Greengrass nucleus ZIP (`greengrass-nucleus-latest.zip`) ファイルへのフルパス。このフィールドは Greengrass での PreInstalledテストには必要ありません。

> **Note**
>
> IDT for AWS IoT Greengrass でサポートされている Greengrass nucleus のバージョンの詳細については、「for AWS IoT Greengrass V2 の IDT の最新バージョン」を参照してください。最新の Greengrass ソフトウェアをダウンロードするには、「AWS IoT Greengrass ソフトウェアをダウンロードする」を参照してください。

`PreInstalled`

この機能は、Linux デバイスでのみ IDT v4.5.8 以降で利用できます。

(オプション) 値が *`yes`* の場合、IDT は `InstallationDirRootOnDevice` に記載されているパスを Greengrass がインストールされているディレクトリとみなします。

デバイスで Greengrass をインストールする方法の詳細については、「自動リソースプロビジョニング機能を備えた AWS IoT Greengrass Core ソフトウェアをインストール」を参照してください。手動プロビジョニングでインストールする 場合は、AWS IoT のモノ を手動で作成するときに「AWS IoT のモノを新規または既存のモノグループに追加する」手順を含めてください。IDT では、モノとモノのグループがインストールのセットアップ中に作成されると想定しています。これらの値が `effectiveConfig.yaml` ファイルに反映されていることを確認してください。IDT は `<InstallationDirRootOnDevice>/config/effectiveConfig.yaml` の下の `effectiveConfig.yaml` ファイルをチェックします。

HSM でテストを実行する場合は、`aws.greengrass.crypto.Pkcs11Provider` フィールドが `effectiveConfig.yaml` で更新されていることを確認してください。

`GreengrassV2TokenExchangeRole`

(オプション) テスト対象のデバイスが AWS リソースとやり取りすることを想定した、トークン交換ロールとして使用するカスタム IAM ロール。

> **Note**
>
> IDT は、テストの実行中にデフォルトのトークン交換ロールを作成する代わりに、このカスタム IAM ロールを使用します。カスタムロールを使用する場合は、[IAM permissions

for the test user] (テストユーザーの IAM アクセス許可) を更新して、ユーザーが IAM ロールとポリシーを作成および削除できるようにする `iamResourcesUpdate` ステートメントを除外できます。

トークン交換ロールとしてカスタム IAM ロールの作成方法の詳細については、「カスタムトークン交換ロールを設定する」を参照してください。

`hsm`

この機能は IDT v4.5.1 以降で利用できます。

(オプション) AWS IoT Greengrass ハードウェアセキュリティモジュール (HSM) でテストするための設定情報。それ以外の場合は、`hsm` プロパティを省略する必要があります。詳細については、「ハードウェアセキュリティ統合」を参照してください。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

> **Warning**
>
> ハードウェアセキュリティモジュールが IDT と別のシステム間で共有されている場合、HSM 設定は機密データとみなされる場合があります。このような状況では、これらの設定値を Parameter Store SecureString パラメータに保存し、テスト実行中に取得するように IDT AWS を設定することで、これらの設定値をプレーンテキストで保護することを避けることができます。詳細については、「???」を参照してください。

`hsm.greengrassPkcsPluginJar`

IDT ホストマシンにダウンロードする [PKCS#11 provider component] (PKCS#11 プロバイダコンポーネント) のフルパス。AWS IoT Greengrass はこのコンポーネントを JAR ファイルとして提供します。このファイルをダウンロードし、インストール時にプロビジョニングプラグインとして指定できます。次の URL としてコンポーネントの JAR ファイルの最新バージョンをダウンロードできます: https://d2s8p88vqu9w66.cloudfront.net/releases/Pkcs11Provider/aws.greengrass.crypto.Pkcs11Provider-latest.jar。

`hsm.pkcs11ProviderLibrary`

HSM とやり取りするためのハードウェアセキュリティモジュール (HSM) ベンダーによって提供される PKCS#11 ライブラリへのフルパス。

`hsm.slotId`

キーと証明書をロードする HSM スロットの識別に使用されるスロット ID。

`hsm.slotLabel`

キーと証明書をロードする HSM スロットの識別に使用されるスロットラベル。

`hsm.slotUserPin`

IDT が AWS IoT Greengrass Core ソフトウェアを HSM に認証するために使用するユーザー PIN。

> **Note**
>
> セキュリティのベストプラクティスとして、実稼働デバイスで同じユーザー PIN を使用しないでください。

`hsm.keyLabel`

ハードウェアモジュールでキーを識別するために使用されるラベル。キーと証明書の両方で同じキーラベルを使用する必要があります。

`hsm.preloadedCertificateArn`

AWS IoT クラウドのデバイス証明書の Amazon リソースネーム (ARN)。

HSM のキーを使用してこの証明書を事前に生成し、HSM にインポートして、AWS IoT クラウドにアップロードしておく必要があります。証明書の生成とインポートの詳細については、「HSM のドキュメント」を参照してください。

証明書は、config.json. で指定したのと同じアカウントとリージョンにアップロードする必要があります。証明書を AWS IoT にアップロードする方法の詳細については、「AWS IoT デベロッパーガイド」の「クライアント証明書を手動で登録する」を参照してください。

`hsm.rootCAPath`

(オプション) IDT ホストマシンで、証明書に署名したルート認証局 (CA) へのフルパス。これは、作成された HSM の証明書が Amazon ルート CA によって署名されていない場合に必要です。

AWS Parameter Store から設定をフェッチする

AWS IoT Device Tester (IDT) には、AWSSystems Manager Parameter StoreAWS から設定値をフェッチするオプション機能が含まれています。Parameter Store を使用することで、設定を暗号化して安全に保存できます。設定すると、IDT は、`userdata.json` ファイル内にプレーンテキストでパラメータを格納する代わりに、AWS Parameter Store からパラメータをフェッチできます。これは、パスワード、PIN、その他の秘密など、暗号化して保存する必要がある機密データにとって有益です。

1. この機能を使用するには、IDT ユーザーの作成に使用されるアクセス許可を更新して、IDT が使用するように設定されているパラメータに対する GetParameter アクションを許可する必要があります。IDT ユーザーに追加できる許可ステートメントの例を以下に示します。詳細については、「AWS Systems Manager ユーザーガイド」を参照してください。

   ```
   {
          "Sid":"parameterStoreResources",
          "Effect": "Allow",
           "Action": [
               "ssm:GetParameter"
           ],
           "Resource": "arn:aws:ssm:*:*:parameter/IDT*"
   }
   ```

   上記の許可は、ワイルドカード文字 * を使用することによって、名前が IDT で始まるすべてのパラメータをフェッチすることを許可するように設定されています。使用しているパラメータの名前に基づいて、設定されたパラメータを IDT がフェッチできるように、必要に応じてこれをカスタマイズする必要があります。

2. AWS Parameter Store 内に設定値を保存する必要があります。これは、AWS コンソールまたは AWS CLI から実行できます。AWSParameter Store では、暗号化されたストレージまたは暗号化されていないストレージを選択できます。シークレット、パスワード、ピンなどの機密値を保存するには、 のパラメータタイプである暗号化オプションを使用する必要がありますSecureString。AWS CLI を使用してパラメータをアップロードするには、次のコマンドを使用します。

   ```
   aws ssm put-parameter --name IDT-example-name --value IDT-example-value --type
    SecureString
   ```

次のコマンドを実行して、パラメータが保存されていることを確認できます。(オプション) --with-decryptionフラグを使用して、復号化されたSecureString パラメータを取得します。

```
aws ssm get-parameter --name IDT-example-name
```

AWS CLI を使用すると、現在の CLI ユーザーの AWS リージョンにパラメータがアップロードされ、IDT は config.json で設定されたリージョンからパラメータをフェッチします。AWS CLI からリージョンを確認するには、次を使用します。

```
aws configure get region
```

3. AWS クラウドで設定値を取得すると、IDT 設定内の任意の値を更新して、AWS クラウドからフェッチできるようになります。これを実行するには、フォーム {{AWS.Parameter.parameter_name}} の IDT 設定でプレースホルダーを使用して、AWS Parameter Store からその名前でパラメータをフェッチします。

   例えば、ステップ 2 の IDT-example-name パラメータを HSM 設定で HSM keyLabel として使用する場合は、userdata.json を次のように更新できます。

```
"hsm": {
        "keyLabel": "{{AWS.Parameter.IDT-example-name}}",
        [...]
    }
```

   IDT は、ステップ 2 で IDT-example-value に設定されたこのパラメータの値を実行時にフェッチします。この設定は "keyLabel": "IDT-example-value" の設定に似ていますが、その値が暗号化されて AWS クラウドに保存される点で異なります。

AWS IoT Greengrass 資格 Suite の実行

必要な設定を定義したら、テストを開始できます。完全なテストスイートのランタイムはハードウェアによって異なります。参考までに、Raspberry Pi 3B で完全なテストスイートを完了するには約 30 分かかります。

次の run-suite コマンドを使用して、一連のテストを実行します。

```
devicetester_[linux | mac | win]_x86-64 run-suite  \\
    --suite-id suite-id  \\
```

```
    --group-id group-id  \\
    --pool-id your-device-pool \\
    --test-id test-id  \\
    --update-idt y|n  \\
    --userdata userdata.json
```

すべてオプションは任意です。例えば、device.json ファイルに定義されている同一のデバイスのセットであるデバイスプールが 1 つしかない場合は、pool-id を省略できます。または、tests フォルダ内の最新のテストスイートのバージョンを実行する場合は、suite-id を省略できます。

> **Note**
>
> IDT は、新しいテストスイートのバージョンをオンラインで入手できるかどうかを尋ねるプロンプトを表示します。詳細については、「the section called “テストスイートのバージョン”」を参照してください。

資格スイートを実行するためのコマンド例

以下のコマンドラインの例では、デバイスプールに対して適格性確認テストを実行する方法を示します。run-suite およびその他の IDT コマンドの詳細については、「the section called “IDT コマンド”」を参照してください。

指定のテストスイートのテストグループをすべて実行するには、次のコマンドを使用します。list-suites コマンドは、tests フォルダにあるテストスイートを一覧表示します。

```
devicetester_[linux | mac | win]_x86-64 run-suite \
    --suite-id GGV2Q_1.0.0 \
    --pool-id <pool-id> \
    --userdata userdata.json
```

テストスイートの特定のテストグループを実行するには、次のコマンドを使用します。list-groups コマンドは、テストスイートのテストグループを一覧表示します。

```
devicetester_[linux | mac | win]_x86-64 run-suite \
    --suite-id GGV2Q_1.0.0 \
    --group-id <group-id> \
    --pool-id <pool-id> \
    --userdata userdata.json
```

テストグループの特定のテストケースを実行するには、次のコマンドを使用します。

```
devicetester_[linux | mac | win]_x86-64 run-suite \
    --group-id <group-id> \
    --test-id <test-id> \
    --userdata userdata.json
```

テストグループの複数のテストケースを実行するには、次のコマンドを使用します。

```
devicetester_[linux | mac | win]_x86-64 run-suite \
    --group-id <group-id> \
    --test-id <test-id1>,<test-id2>
    --userdata userdata.json
```

テストグループのすべてのテストケースを一覧表示するには、次のコマンドを使用します。

```
devicetester_[linux | mac | win]_x86-64 list-test-cases --group-id <group-id>
```

テストグループの依存関係を正しい順序で実行する、完全な認定テストスイートを実行することをお勧めします。特定のテストグループを実行することを選択した場合、関連するテストグループを実行する前に、まず依存関係チェッカーテストグループを実行し、すべての Greengrass の依存関係がインストールされていることを確認することをお勧めします。例:

- コア資格テストグループを実行する前に `coredependencies` を実行します。

AWS IoT Greengrass V2 コマンドの IDT

IDT コマンドは、*<device-tester-extract-location>*/bin ディレクトリにあります。テストスイートを実行するには、次の形式でコマンドを提供します。

`help`

指定されたコマンドに関する情報を一覧表示します。

`list-groups`

特定のテストスイート内のグループを一覧表示します。

`list-suites`

使用可能なテストスイートを一覧表示します。

`list-supported-products`

サポート対象製品 (この場合は AWS IoT Greengrass バージョン) と最新の IDT バージョンのテストスイートのバージョンを一覧表示します。

`list-test-cases`

指定したテストグループのテストケースを一覧表示します。次のオプションがサポートされています。

- `group-id`。検索するテストグループ。このオプションは必須で、1 つのグループを指定する必要があります。

`run-suite`

デバイスプールに対してテストスイートを実行します。サポートされているオプションの一部は以下のとおりです。

- `suite-id`。実行するテストスイートのバージョン。指定しない場合、IDT は `tests` フォルダにある最新バージョンを使用します。
- `group-id`。実行するテストグループ (カンマ区切りリストとして)。指定しない場合、IDT は、`device.json` で設定された設定に応じて、テストスイートのすべての適切なテストグループを実行します。IDT は、設定した設定に基づいてデバイスがサポートしていないテストグループを実行しません。(これらのテストグループが `group-id` リストで指定されている場合でも同様)。
- `test-id`。実行するテストケース (カンマ区切りリストとして)。指定した場合は、`group-id` は 1 つのグループを指定する必要があります。
- `pool-id`。テストするデバイスプール。`device.json` ファイルに複数のデバイスプールが定義されている場合は、プールを指定する必要があります。
- `stop-on-first-failure`。最初に障害で実行を停止するように IDT を設定します。指定されたテストグループをデバッグする場合は、このオプションを `group-id` とともに使用します。完全テストスイートを実行して認定レポートを生成する場合は、このオプションを使用しないでください。
- `update-idt`。 IDT を更新するプロンプトの応答を設定します。IDT が新しいバージョンがあることを検出した場合、Y 応答はテストの実行を停止します。N 応答は、テストの実行を継続します。
- `userdata`。テストアーティファクトパスに関する情報を含む `userdata.json` ファイルへのフルパス。このオプションは、`run-suite` コマンドに必要です。`userdata.json` ファイルは *`devicetester_extract_location`*/devicetester_ggv2_*`[win|mac|linux]`*/configs/ ディレクトリに配置される必要があります。

`run-suite` オプションの詳細については、次の `help` オプションを使用してください。

```
devicetester_[linux | mac | win]_x86-64 run-suite -h
```

結果とログを理解する

このセクションでは、IDT の結果レポートとログを表示し、解釈する方法について説明します。

エラーをトラブルシューティングする方法については、「IDT for AWS IoT Greengrass V2 のトラブルシューティング」を参照してください。

結果の表示

実行中、IDT はコンソール、ログファイル、テストレポートにエラーを書き込みます。IDT で適格性テストスイートを実行すると、2 つのレポートが生成されます。これらのレポートは *<device-tester-extract-location>*`/results/`*<execution-id>*`/` にあります。両レポートとも、適格性確認テストスイートを実行した結果をキャプチャします。

`awsiotdevicetester_report.xml` は、AWS Partner Device Catalog にデバイスを記載するために AWS に提出する適格性確認テストレポートです。レポートには、次の要素が含まれます。

- IDT バージョン。
- テストされた AWS IoT Greengrass のバージョン。
- `device.json` ファイルで指定されている SKU とデバイスプール。
- `device.json` ファイルで指定されているデバイスプールの機能。
- テスト結果の概要の集計。
- デバイスの機能 (ローカル リソース アクセス、シャドウ、MQTT など) に基づいてテストしたライブラリ別のテスト結果の内訳。

`GGV2Q_Result.xml` レポートは JUnit XML 形式です。Jenkins、Bamboo などのように継続的な統合 (CI) と継続的なデプロイ (CD) のプラットフォームに統合することができます。レポートには、次の要素が含まれます。

- テスト結果の概要を集計したもの。
- テストされた AWS IoT Greengrass 機能別のテスト結果の内訳。

AWS IoT Device Tester 結果の解釈

awsiotdevicetester_report.xml または awsiotdevicetester_report.xml のレポートセクションには、実行されたテストとその結果が一覧表示されます。

最初の XML タグ <testsuites> には、テスト実行の概要が含まれています。例:

```
<testsuites name="GGQ results" time="2299" tests="28" failures="0" errors="0"
 disabled="0">
```

<testsuites> タグで使用される属性

name

テストスイートの名前。

time

適格性確認スイートの実行所要時間 (秒単位)。

tests

実行されたテスト数。

failures

実行されたテストのうち、合格しなかったものの数。

errors

IDT が実行できなかったテスト数。

disabled

この属性を無視します。それは使用されていません。

awsiotdevicetester_report.xml ファイルには、テスト対象の製品および一連のテストの実行後に検証された製品機能に関する情報を含む <awsproduct> タグが含まれています。

<awsproduct> タグで使用される属性

name

テスト対象の製品の名前。

`version`

テスト対象の製品のバージョン。

`features`

検証された機能です。`required` としてマークされている機能については、資格を得るためにボードを提出するために必要です。次のスニペットは、この情報が `awsiotdevicetester_report.xml` ファイルで表示される方法を示します。

```
<name="aws-iot-greengrass-v2-core" value="supported" type="required"></feature>
```

必要な機能に対してテストに障害やエラーがない場合、そのデバイスは AWS IoT Greengrass を実行するための技術的要件を満たしており、AWS IoT サービスとの相互運用が可能です。AWS Partner Device Catalog にデバイスを記載する場合、認定の証拠としてこのレポートを使用できます。

テストに障害やエラーが発生した場合は、`<testsuites>` XML タグを確認することで、障害の生じたテストを特定できます。`<testsuites>` タグ内の `<testsuite>` XML タグは、テストグループのテスト結果の要約を示します。例:

```
<testsuite name="combination" package="" tests="1" failures="0" time="161" disabled="0"
 errors="0" skipped="0">
```

形式は `<testsuites>` タグと似ていますが、使用されていないため無視できる `skipped` という属性があります。各 `<testsuite>` XML タグには、テストグループに実行されたテストごとに `<testcase>` タグがあります。例:

```
<testcase classname="Security Combination (IPD + DCM) Test Context" name="Security
 Combination IP Change Tests sec4_test_1: Should rotate server cert when IPD disabled
 and following changes are made:Add CIS conn info and Add another CIS conn info"
 attempts="1"></testcase>>
```

`<testcase>` タグで使用される属性

`name`

テストの名前。

`attempts`

IDT がテストケースを実行した回数。

テストに障害やエラーが発生した場合、<failure> タグまたは <error> タグがトラブルシューティングのための情報とともに <testcase> タグに追加されます。例:

```
<testcase classname="mcu.Full_MQTT" name="AFQP_MQTT_Connect_HappyCase" attempts="1">
 <failure type="Failure">Reason for the test failure</failure>
 <error>Reason for the test execution error</error>
</testcase>
```

ログの表示

IDT は、*<devicetester-extract-location>*/results/*<execution-id>*/logs のテスト実行によってログを生成します。2 組のログが生成されます。

`test_manager.log`

AWS IoT Device Tester (例えば、設定、テストシーケンス、レポート生成に関連したログ) の Test Manager コンポーネントから生成されたログ。

<test-case-id>`.log (for example, lambdaDeploymentTest.log)`

テストグループ内のテストケースのログ (テスト対象デバイスのログも含む)。IDT v4.2.0 以降、IDT は各テストケースのテストログを *<devicetester-extract-location>*/results/*<execution-id>*/logs/*<test-group-id>/* ディレクトリ内の別々な *<test-case-id>* フォルダにグループ化します。

IDT を使用して独自のテストスイートを開発および実行する

IDT v4.0.1 以降、IDT for AWS IoT Greengrass V2 では、標準化された構成設定および結果形式と、デバイスやデバイスソフトウェア用のカスタムテストスイートを開発できるテストスイート環境が統合されています。独自の内部検証用のカスタムテストを追加したり、デバイス検証のためにこれらのテストを顧客に提供したりできます。

IDT を使用してカスタムテストスイートを開発および実行するには、次の手順を実行します。

カスタムテストスイートを開発するには

- テストする Greengrass デバイス用のカスタムテストロジックを使用して、テストスイートを作成します。
- IDT と作成したカスタムテストスイートをテストの実行者に提供します。作成したテストスイートの構成設定に関する情報も提供します。

カスタムテストスイートを実行するには

- テストするデバイスをセットアップします。
- 使用するテストスイートに必要な構成設定を実装します。
- IDT を使用して、カスタムテストスイートを実行します。
- IDT によって実行されたテストのテスト結果と実行ログを表示します。

最新バージョンの AWS IoT Device Tester for AWS IoT Greengrass のダウンロード

IDT の最新バージョンをダウンロードし、読み取り/書き込みアクセス許可を持つファイルシステム上の場所にソフトウェアを抽出します (*#device-tester-extract-location#*)。

> **Note**
>
> 複数のユーザーが NFS ディレクトリや Windows ネットワーク共有フォルダなどの共有場所から IDT を実行することはお勧めしません。IDT パッケージをローカルドライブに展開し、ローカルワークステーションで IDT バイナリを実行することをお勧めします。
> Windows では、パスの長さは 260 文字に制限されています。Windows を使用している場合は、パスが 260 文字以内になるようにして、IDT をルートディレクトリ (`C:\` または `D:\` など) に展開します。

テストスイート作成ワークフロー

テストスイートは 3 つのタイプのファイルで設定されます。

- IDT にテストスイートの実行方法に関する情報を提供する設定ファイル。
- IDT がテストケースの実行に使用するテスト実行可能ファイル。
- テストの実行に必要な追加のファイル。

カスタム IDT テストを作成するには、次の基本的な手順を実行します。

1. テストスイート用の設定ファイルを作成します。
2. テストスイート用のテストロジックが含まれるテストケース実行可能ファイルを作成します。
3. テストスイートを実行するためにテストの実行者に必要な設定情報を検証し、文書化します。
4. IDT が予想通りにテストスイートを実行し、テスト結果を生成できることを確認します。

サンプルカスタムスイートを迅速に構築して実行するには、チュートリアル: サンプル IDT テストスイートを構築して実行するの手順に従ってください。

Python でカスタムテストスイートの作成を開始するには、チュートリアル: シンプルな IDT テストスイートの開発を参照してください。

チュートリアル: サンプル IDT テストスイートを構築して実行する

AWS IoT Device Tester ダウンロードには、サンプルテストスイートのソースコードが含まれています。サンプルテストスイートを構築して実行するこのチュートリアルを完了すると、IDT for AWS IoT Greengrass を使用してカスタムテストスイートを実行する方法を理解することができます。

このチュートリアルでは、次の手順を実行します。

1. サンプルテストスイートを構築する
2. IDT を使用してサンプルテストスイートを実行する

前提条件

このチュートリアルを完了するには、以下が必要です。

- ホストコンピュータの要件
  - AWS IoT Device Tester の最新バージョン
  - Python 3.7 以降

    コンピュータにインストールされている Python のバージョンを確認するには、次のコマンドを実行します。

    ```
    python3 --version
    ```

Windows で、このコマンドを使用してエラーが返された場合は、代わりに `python --version` を使用してください。返されたバージョン番号が 3.7 以上の場合は、Powershell ターミナルで次のコマンドを実行し、`python3` を `python` コマンドのエイリアスとして設定します。

```
Set-Alias -Name "python3" -Value "python"
```

バージョン情報が返されない場合や、バージョン番号が 3.7 未満 の場合は、Python のダウンロードの手順に従って Python 3.7 以上をインストールしてください。詳細については、Python のドキュメントを参照してください。

- urllib3

  `urllib3` が正しくインストールされていることを確認するには、次のコマンドを実行します。

  ```
  python3 -c 'import urllib3'
  ```

  `urllib3` がインストールされていない場合は、次のコマンドを実行してインストールします。

  ```
  python3 -m pip install urllib3
  ```

- デバイスの要件
  - Linux オペレーティングシステムが搭載され、ホストコンピュータと同じネットワークにネットワーク接続するデバイス。

    Raspberry Pi OS が搭載された Raspberry Pi を使用することをお勧めします。Raspberry Pi に SSH をセットアップし、リモートから接続できることを確認します。

IDT 用のデバイス情報を設定する

IDT がテストを実行するためのデバイス情報を設定します。次の情報を使用して、*<device-tester-extract-location>*/configs フォルダに含まれている device.json テンプレートを更新する必要があります。

```
[
  {
    "id": "pool",
    "sku": "N/A",
```

```
    "devices": [
      {
        "id": "<device-id>",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": "<port>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
]
```

devices オブジェクトで、次の情報を指定します。

id

自分のデバイスのユーザー定義の一意の識別子。

connectivity.ip

自分のデバイスの IP アドレス。

connectivity.port

オプションです。デバイスへの SSH 接続に使用するポート番号。

connectivity.auth

接続の認証情報。

このプロパティは、connectivity.protocol が ssh に設定されている場合にのみ適用されます。

connectivity.auth.method

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- pki
- password

`connectivity.auth.credentials`

認証に使用される認証情報。

`connectivity.auth.credentials.user`

デバイスへのサインインに使用するユーザー名。

`connectivity.auth.credentials.privKeyPath`

デバイスへのサインインに使用するプライベートキーへの完全パス。

この値は、`connectivity.auth.method` が `pki` に設定されている場合にのみ適用されます。

`devices.connectivity.auth.credentials.password`

自分のデバイスにサインインするためのパスワード。

この値は、`connectivity.auth.method` が `password` に設定されている場合にのみ適用されます。

> **Note**
>
> `method` が `pki` に設定されている場合のみ `privKeyPath` を指定します。
> `method` が `password` に設定されている場合のみ `password` を指定します。

サンプルテストスイートを構築する

<device-tester-extract-location>/samples/python フォルダには、サンプル設定ファイル、ソースコード、および提供されたビルドスクリプトを使用してテストスイートに結合できる IDT クライアント SDK が含まれています。次のディレクトリツリーは、これらのサンプルファイルの場所を示しています。

```
<device-tester-extract-location>
### ...
```

```
### tests
### samples
#   ### ...
#   ### python
#       ### configuration
#       ### src
#       ### build-scripts
#           ### build.sh
#           ### build.ps1
### sdks
    ### ...
    ### python
        ### idt_client
```

テストスイートを構築するには、ホストコンピュータで次のコマンドを実行します。

Windows

```
cd <device-tester-extract-location>/samples/python/build-scripts
./build.ps1
```

Linux, macOS, or UNIX

```
cd <device-tester-extract-location>/samples/python/build-scripts
./build.sh
```

これにより、*<device-tester-extract-location>*/tests フォルダ内の IDTSampleSuitePython_1.0.0 フォルダにサンプルテストスイートが作成されます。IDTSampleSuitePython_1.0.0 フォルダのファイルを確認して、サンプルテストスイートの構造を理解し、テストケースの実行可能ファイルとテスト設定 JSON ファイルのさまざまな例を参照してください。

> **Note**
>
> サンプルテストスイートには python ソースコードが含まれます。テストスイートのコードには機密情報を入力しないでください。

次のステップ: IDT を使用して、作成したサンプルテストスイートを実行します。

IDT を使用してサンプルテストスイートを実行する

サンプルテストスイートを実行するには、ホストコンピュータで次のコマンドを実行します。

```
cd <device-tester-extract-location>/bin
./devicetester_[linux | mac | win_x86-64] run-suite --suite-id IDTSampleSuitePython
```

IDT はサンプルテストスイートを実行し、結果をコンソールにストリーミングします。テストの実行が完了すると、次の情報が表示されます。

```
========== Test Summary ==========
Execution Time:         5s
Tests Completed:        4
Tests Passed:           4
Tests Failed:           0
Tests Skipped:          0
----------------------------------
Test Groups:
    sample_group:       PASSED
----------------------------------
Path to IoT Device Tester Report: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/logs
Path to Aggregated JUnit Report: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/IDTSampleSuitePython_Report.xml
```

トラブルシューティング

次の情報は、チュートリアルの実行に関連する問題の解決に役立ちます。

テストケースが正常に実行されない

テストが正常に実行されない場合、IDT はエラーログをコンソールにストリーミングします。このログはテスト実行のトラブルシューティングに役立ちます。このチュートリアルのすべての前提条件を満たしていることを確認してください。

テスト対象のデバイスに接続できない

以下について確認します。

- device.json ファイルに、正しい IP アドレス、ポート、および認証情報が含まれている。

- ホストコンピュータから SSH 経由でデバイスに接続できる。

チュートリアル: シンプルな IDT テストスイートの開発

テストスイートは、以下を組み合わせたものです。

- テストロジックが含まれるテスト実行可能ファイル
- テストスイートについて記述する設定ファイル

このチュートリアルでは、IDT for AWS IoT Greengrass を使用して、1 つのテストケースを含む Python テストスイートを開発する方法を説明します。このチュートリアルでは、次の手順を実行します。

1. テストスイートディレクトリを作成する
2. 設定ファイルを作成する
3. テストケース実行可能ファイルを作成する
4. テストスイートを実行する

前提条件

このチュートリアルを完了するには、以下が必要です。

- ホストコンピュータの要件
  - AWS IoT Device Tester の最新バージョン
  - Python 3.7 以降

    コンピュータにインストールされている Python のバージョンを確認するには、次のコマンドを実行します。

    ```
    python3 --version
    ```

    Windows で、このコマンドを使用してエラーが返された場合は、代わりに `python --version` を使用してください。返されたバージョン番号が 3.7 以上の場合は、Powershell ターミナルで次のコマンドを実行し、`python3` を `python` コマンドのエイリアスとして設定します。

```
Set-Alias -Name "python3" -Value "python"
```

バージョン情報が返されない場合や、バージョン番号が 3.7 未満 の場合は、Python のダウンロードの手順に従って Python 3.7 以上をインストールしてください。詳細については、Python のドキュメントを参照してください。

- urllib3

  `urllib3` が正しくインストールされていることを確認するには、次のコマンドを実行します。

  ```
  python3 -c 'import urllib3'
  ```

  `urllib3` がインストールされていない場合は、次のコマンドを実行してインストールします。

  ```
  python3 -m pip install urllib3
  ```

- デバイスの要件
  - Linux オペレーティングシステムが搭載され、ホストコンピュータと同じネットワークにネットワーク接続するデバイス。

    Raspberry Pi OS が搭載された Raspberry Pi を使用することをお勧めします。Raspberry Pi に SSH をセットアップし、リモートから接続できることを確認します。

テストスイートディレクトリを作成する

IDT は、テストケースを、各テストスイート内のテストグループに論理的に分離します。各テストケースはテストグループ内に存在する必要があります。このチュートリアルでは、`MyTestSuite_1.0.0` という名前のフォルダを作成し、このフォルダ内に次のディレクトリツリーを作成します。

```
MyTestSuite_1.0.0
### suite
    ### myTestGroup
        ### myTestCase
```

設定ファイルを作成する

テストスイートには、次の必須の設定ファイルが含まれている必要があります。

必要な設定ファイル

`suite.json`

テストスイートに関する情報が含まれています。suite.json を設定する を参照してください。

`group.json`

テストグループに関する情報が含まれています。テストスイート内のテストグループごとに `group.json` ファイルを作成する必要があります。group.json を設定する を参照してください。

`test.json`

テストケースに関する情報が含まれています。テストスイート内のテストケースごとに `test.json` ファイルを作成する必要があります。test.json を設定する を参照してください。

1. `MyTestSuite_1.0.0/suite` フォルダで、次の構造の `suite.json` ファイルを作成します。

```
{
    "id": "MyTestSuite_1.0.0",
    "title": "My Test Suite",
    "details": "This is my test suite.",
    "userDataRequired": false
}
```

2. `MyTestSuite_1.0.0/myTestGroup` フォルダで、次の構造の `group.json` ファイルを作成します。

```
{
    "id": "MyTestGroup",
    "title": "My Test Group",
    "details": "This is my test group.",
    "optional": false
}
```

3. `MyTestSuite_1.0.0/myTestGroup/myTestCase` フォルダで、次の構造の `test.json` ファイルを作成します。

```
{
    "id": "MyTestCase",
    "title": "My Test Case",
    "details": "This is my test case.",
    "execution": {
```

```
        "timeout": 300000,
        "linux": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        },
        "mac": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        },
        "win": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        }
    }
}
```

MyTestSuite_1.0.0 フォルダのディレクトリツリーは次のようになります。

```
MyTestSuite_1.0.0
### suite
    ### suite.json
    ### myTestGroup
        ### group.json
        ### myTestCase
            ### test.json
```

IDT クライアント SDK を入手する

IDT がテスト対象のデバイスとやり取りし、テスト結果をレポートできるようにするには、IDT クライアント SDK を使用します。このチュートリアルでは、Python バージョンの SDK を使用します。

<device-tester-extract-location>/sdks/python/ フォルダから、idt_client フォルダを自分の MyTestSuite_1.0.0/suite/myTestGroup/myTestCase フォルダにコピーします。

SDK が正常にコピーされたことを確認するには、次のコマンドを実行します。

```
cd MyTestSuite_1.0.0/suite/myTestGroup/myTestCase
python3 -c 'import idt_client'
```

テストケース実行可能ファイルを作成する

テストケース実行可能ファイルには、実行するテストロジックが含まれています。テストスイートには、複数のテストケース実行可能ファイルを含めることができます。このチュートリアルでは、テストケース実行可能ファイルを 1 つだけ作成します。

1. テストスイートファイルを作成します。

   MyTestSuite_1.0.0/suite/myTestGroup/myTestCase フォルダで、次の内容の myTestCase.py ファイルを作成します。

   ```
   from idt_client import *

   def main():
       # Use the client SDK to communicate with IDT
       client = Client()

   if __name__ == "__main__":
       main()
   ```

2. クライアント SDK 関数を使用して、自分の myTestCase.py ファイルに次のテストロジックを追加します。

   a. テスト対象のデバイスで SSH コマンドを実行します。

   ```
   from idt_client import *

   def main():
       # Use the client SDK to communicate with IDT
       client = Client()

       # Create an execute on device request
       exec_req = ExecuteOnDeviceRequest(ExecuteOnDeviceCommand("echo 'hello
    world'"))

       # Run the command
       exec_resp = client.execute_on_device(exec_req)

       # Print the standard output
   ```

```
    print(exec_resp.stdout)

if __name__ == "__main__":
    main()
```

b. テスト結果を IDT に送信します。

```
from idt_client import *

def main():
    # Use the client SDK to communicate with IDT
    client = Client()

    # Create an execute on device request
    exec_req = ExecuteOnDeviceRequest(ExecuteOnDeviceCommand("echo 'hello
 world'"))

    # Run the command
    exec_resp = client.execute_on_device(exec_req)

    # Print the standard output
    print(exec_resp.stdout)

    # Create a send result request
    sr_req = SendResultRequest(TestResult(passed=True))

    # Send the result
    client.send_result(sr_req)

if __name__ == "__main__":
    main()
```

IDT 用のデバイス情報を設定する

IDT がテストを実行するためのデバイス情報を設定します。次の情報を使用して、*<device-tester-extract-location>*/configs フォルダに含まれている device.json テンプレートを更新する必要があります。

```
[
  {
    "id": "pool",
    "sku": "N/A",
```

```
    "devices": [
      {
        "id": "<device-id>",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": "<port>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
]
```

devices オブジェクトで、次の情報を指定します。

`id`

自分のデバイスのユーザー定義の一意の識別子。

`connectivity.ip`

自分のデバイスの IP アドレス。

`connectivity.port`

オプションです。デバイスへの SSH 接続に使用するポート番号。

`connectivity.auth`

接続の認証情報。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.auth.method`

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- pki
- password

connectivity.auth.credentials

認証に使用される認証情報。

connectivity.auth.credentials.user

デバイスへのサインインに使用するユーザー名。

connectivity.auth.credentials.privKeyPath

デバイスへのサインインに使用するプライベートキーへの完全パス。

この値は、connectivity.auth.method が pki に設定されている場合にのみ適用されます。

devices.connectivity.auth.credentials.password

自分のデバイスにサインインするためのパスワード。

この値は、connectivity.auth.method が password に設定されている場合にのみ適用されます。

> **Note**
>
> method が pki に設定されている場合のみ privKeyPath を指定します。
> method が password に設定されている場合のみ password を指定します。

テストスイートを実行する

テストスイートを作成したら、テストスイートが期待どおりに機能することを確認します。そのために、次の手順に従って、既存のデバイスプールを使用してテストスイートを実行します。

1. 自分の MyTestSuite_1.0.0 フォルダを *<device-tester-extract-location>*/tests にコピーします。
2. 以下のコマンドを実行します。

```
cd <device-tester-extract-location>/bin
./devicetester_[linux | mac | win_x86-64] run-suite --suite-id MyTestSuite
```

IDT はテストスイートを実行し、結果をコンソールにストリーミングします。テストの実行が完了すると、次の情報が表示されます。

```
time="2020-10-19T09:24:47-07:00" level=info msg=Using pool: pool
time="2020-10-19T09:24:47-07:00" level=info msg=Using test suite "MyTestSuite_1.0.0"
 for execution
time="2020-10-19T09:24:47-07:00" level=info msg=b'hello world\n'
 suiteId=MyTestSuite groupId=myTestGroup testCaseId=myTestCase deviceId=my-device
 executionId=9a52f362-1227-11eb-86c9-8c8590419f30
time="2020-10-19T09:24:47-07:00" level=info msg=All tests finished.
 executionId=9a52f362-1227-11eb-86c9-8c8590419f30
time="2020-10-19T09:24:48-07:00" level=info msg=

========== Test Summary ==========
Execution Time:         1s
Tests Completed:        1
Tests Passed:           1
Tests Failed:           0
Tests Skipped:          0
----------------------------------
Test Groups:
    myTestGroup:        PASSED
----------------------------------
Path to IoT Device Tester Report: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/logs
Path to Aggregated JUnit Report: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/MyTestSuite_Report.xml
```

トラブルシューティング

次の情報は、チュートリアルの実行に関連する問題の解決に役立ちます。

テストケースが正常に実行されない

テストが正常に実行されない場合、IDT はエラーログをコンソールにストリーミングします。このログはテスト実行のトラブルシューティングに役立ちます。エラーログを確認する前に、次の点を確認してください。

- IDT クライアント SDK が、このステップで説明された通りの正しいフォルダにある。
- このチュートリアルのすべての前提条件を満たしている。

テスト対象のデバイスに接続できない

以下について確認します。

- device.json ファイルに、正しい IP アドレス、ポート、および認証情報が含まれている。
- ホストコンピュータから SSH 経由でデバイスに接続できる。

IDT テストスイート設定ファイルを作成する

このセクションでは、カスタムテストスイートの作成時、これに含める設定ファイルを作成する際の形式について説明します。

必要な設定ファイル

`suite.json`

テストスイートに関する情報が含まれています。suite.json を設定する を参照してください。

`group.json`

テストグループに関する情報が含まれています。テストスイート内のテストグループごとに group.json ファイルを作成する必要があります。group.json を設定する を参照してください。

`test.json`

テストケースに関する情報が含まれています。テストスイート内のテストケースごとに test.json ファイルを作成する必要があります。test.json を設定する を参照してください。

オプションの設定ファイル

`test_orchestrator.yaml` または `state_machine.json`

IDT がテストスイートを実行するときのテストの実行方法を定義します。「test_orchestrator.yaml を設定する」を参照してください。

> **Note**
>
> IDT v4.5.1 以降では、テストワークフローの定義に test_orchestrator.yaml ファイルを使用します。IDT のそれより前のバージョンでは、state_machine.jsonファイルを開きます。ステートマシンの詳細については、「IDT ステートマシンを構成する」を参照してください。

`userdata_schema.json`

テストの実行者が構成設定に含めることができる `userdata.json` ファイルのスキーマを定義します。`userdata.json` ファイルは、テストの実行に必要であるものの、`device.json` ファイルに含まれていない、追加の設定情報用に使用します。userdata_schema.json を設定する を参照してください。

設定ファイルは、以下に示すように *<custom-test-suite-folder>* に配置します。

```
<custom-test-suite-folder>
### suite
    ### suite.json
    ### test_orchestrator.yaml
    ### userdata_schema.json
    ### <test-group-folder>
        ### group.json
        ### <test-case-folder>
            ### test.json
```

suite.json を設定する

`suite.json` ファイルは、環境変数を設定し、テストスイートの実行にユーザーデータが必要かどうかを決定します。以下のテンプレートを使用して、*<custom-test-suite-folder>*/suite/`suite.json` ファイルを設定します。

```
{
    "id": "<suite-name>_<suite-version>",
    "title": "<suite-title>",
    "details": "<suite-details>",
    "userDataRequired": true | false,
    "environmentVariables": [
        {
            "key": "<name>",
            "value": "<value>",
        },
        ...
        {
            "key": "<name>",
            "value": "<value>",
        }
    ]
```

```
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`id`

テストスイートの一意のユーザー定義 ID。id の値は、suite.json ファイルが配置されているテストスイートフォルダ名と一致する必要があります。スイート名とスイートのバージョンは、次の要件も満たしている必要があります。

- *<suite-name>* にアンダースコアを含めることはできません。
- *<suite-version>* が *x.x.x* として表されている (x は数字)。

ID は IDT によって生成されるテストレポートに表示されます。

`title`

このテストスイートでテストされる製品または機能のユーザー定義名。この名前は、テストの実行者の IDT CLI に表示されます。

`details`

テストスイートの目的の簡単な説明。

`userDataRequired`

テストの実行者が userdata.json ファイルにカスタム情報を含める必要があるかどうかを定義します。この値を true に設定した場合は、テストスイートフォルダに userdata_schema.json ファイルも含める必要があります。

`environmentVariables`

オプションです。このテストスイートに設定する環境変数の配列。

`environmentVariables.key`

環境変数の名前。

`environmentVariables.value`

環境変数の値。

group.json を設定する

group.json ファイルは、テストグループが必須かオプションかを定義します。以下のテンプレートを使用して、*<custom-test-suite-folder>*/suite/*<test-group>*/group.json ファイルを設定します。

```
{
    "id": "<group-id>",
    "title": "<group-title>",
    "details": "<group-details>",
    "optional": true | false,
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`id`

テストグループの一意のユーザー定義 ID。id の値は、group.json ファイルが配置されているテストグループフォルダの名前と一致する必要があり、アンダースコア (_) は使用できません。ID は IDT によって生成されるテストレポートで使用されます。

`title`

テストグループのわかりやすい名前。この名前は、テストの実行者の IDT CLI に表示されます。

`details`

テストグループの目的の簡単な説明。

`optional`

オプションです。true に設定すると、IDT が必須テストの実行を完了した後に、このテストグループがオプショングループとして表示されます。デフォルト値は false です。

test.json を設定する

test.json ファイルは、テストケースによって使用されるテストケース実行ファイルと環境変数を決定します。テストケース実行可能ファイルの作成の詳細については、IDT テストケース実行可能ファイルを作成するを参照してください。

以下のテンプレートを使用して、*<custom-test-suite-folder>*/suite/*<test-group>*/*<test-case>*/test.json ファイルを設定します。

```
{
    "id": "<test-id>",
    "title": "<test-title>",
    "details": "<test-details>",
    "requireDUT": true | false,
    "requiredResources": [
        {
            "name": "<resource-name>",
            "features": [
                {
                    "name": "<feature-name>",
                    "version": "<feature-version>",
                    "jobSlots": <job-slots>
                }
            ]
        }
    ],
    "execution": {
        "timeout": <timeout>,
        "mac": {
            "cmd": "/path/to/executable",
            "args": [
                "<argument>"
            ],
        },
        "linux": {
            "cmd": "/path/to/executable",
            "args": [
                "<argument>"
            ],
        },
        "win": {
            "cmd": "/path/to/executable",
            "args": [
                "<argument>"
            ]
        }
    },
    "environmentVariables": [
        {
            "key": "<name>",
            "value": "<value>",
        }
```

```
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`id`

テストケースの一意のユーザー定義 ID。`id` の値は、`test.json` ファイルが配置されているテストケースフォルダの名前と一致する必要があり、アンダースコア (_) は使用できません。ID は IDT によって生成されるテストレポートで使用されます。

`title`

テストケースのわかりやすい名前。この名前は、テストの実行者の IDT CLI に表示されます。

`details`

テストケースの目的の簡単な説明。

`requireDUT`

オプションです。このテストの実行にデバイスが必要な場合は `true` に設定します。必要ない場合は `false` に設定します。デフォルト値は `true` です。テストの実行者は、テストの実行に使用するデバイスを `device.json` ファイルに設定します。

`requiredResources`

オプションです。このテストの実行に必要なリソースデバイスに関する情報を指定する配列。

`requiredResources.name`

このテストの実行時にリソースデバイスに与える一意の名前。

`requiredResources.features`

ユーザー定義のリソースデバイス機能の配列。

`requiredResources.features.name`

機能の名前。このデバイスが使用するデバイス機能。この名前は、テストの実行者によって `resource.json` ファイルに指定される機能名に対してマッチングされます。

`requiredResources.features.version`

オプションです。機能のバージョン。この値は、テストの実行者によって `resource.json` ファイルに指定される機能のバージョンに対してマッチングされます。

バージョンが指定されていない場合、機能はチェックされません。機能にバージョン番号が必要ない場合は、このフィールドは空白のままにしてください。

`requiredResources.features.jobSlots`

オプションです。この機能がサポートできる同時テストの数。デフォルト値は、1です。IDT が機能ごとに異なるデバイスを使用する場合は、この値を 1 に設定することをお勧めします。

`execution.timeout`

IDT がテスト実行終了まで待機する時間 (ミリ秒単位)。この値の設定の詳細については、IDT テストケース実行可能ファイルを作成するを参照してください。

`execution.os`

IDT を実行するホストコンピュータのオペレーティングシステムに基づいて実行されるテストケースの実行可能ファイル。サポートされている値は `linux`、`mac`、`win` です。

`execution.os.cmd`

指定されたオペレーティングシステムで実行するテストケース実行可能ファイルへのパス。この場所は、システムパス内に存在する必要があります。

`execution.os.args`

オプションです。テストケースの実行可能ファイルを実行するために指定する引数。

`environmentVariables`

オプションです。このテストケース用に設定された環境変数の配列。

`environmentVariables.key`

環境変数の名前。

`environmentVariables.value`

環境変数の値。

> **Note**
>
> `test.json` ファイルと `suite.json` ファイルに同じ環境変数を設定した場合は、`test.json` ファイルの値が優先されます。

test_orchestrator.yaml を設定する

テストオーケストレーターは、テストスイートの実行フローを制御するコンストラクトです。テストスイートの開始ステートを決定し、ユーザー定義のルールに基づいてステートの移行を管理し、終了ステートに達するまでステートの移行を継続します。

テストスイートにユーザー定義のテストオーケストレーターが含まれていない場合は、IDT によってテストオーケストレーターが生成されます。

デフォルトのテストオーケストレーターには以下の機能があります。

- テストの実行者に、テストスイート全体ではなく、特定のテストグループを選択して実行する機能を提供する。
- 特定のテストグループが選択されていない場合、テストスイート内のすべてのテストグループをランダムな順序で実行する。
- レポートを生成し、各テストグループおよびテストケースのテスト結果を示すコンソールサマリーを出力する。

IDT テストオーケストレーターの機能の詳細については、「IDT テストオーケストレーターを設定する」を参照してください。

userdata_schema.json を設定する

`userdata_schema.json` ファイルは、テストの実行者がユーザーデータを指定するスキーマを決定します。ユーザーデータは、テストスイートが `device.json` ファイルに含まれていない情報を必要とする場合に必要になります。例えば、テストを実行するために、Wi-Fi ネットワークの認証情報、特定のオープンポート、またはユーザーが提供する証明書が必要になる場合があります。この情報は、`userdata` という入力パラメータとして IDT に提供できます。これは、ユーザーが *`<device-tester-extract-location>`*`/config` フォルダに作成する`userdata.json` ファイルの値です。`userdata.json` の形式は、テストケースに含まれている `userdata_schema.json` ファイルに基づきます。

テストの実行者が `userdata.json` ファイルを提供しなければならないことを示す方法

1. `suite.json` ファイルで、`userDataRequired` を `true` に設定します。
2. *`<custom-test-suite-folder>`* で、`userdata_schema.json` ファイルを作成します。
3. `userdata_schema.json` ファイルを編集して、有効な IETF Draft v4 JSON Schema を作成します。

IDT は、テストスイートを実行するときに、このスキーマを自動的に読み込み、テストの実行者によって提供される userdata.json ファイルの検証に使用します。有効な場合、userdata.json ファイルのコンテンツは IDT コンテキストおよび、テストオーケストレーターコンテキストの両方で利用可能になります。

IDT テストオーケストレーターを設定する

IDT v4.5.1 以降、IDT には新しいテストオーケストレーターコンポーネントが追加されています。テストオーケストレーターは、テストスイートの実行フローを制御する IDT コンポーネントで、IDT がすべてのテストを実行した後にテストレポートを生成します。テストオーケストレーターは、ユーザー定義のルールに基づいて、テストの選択とテストの実行順序を決定します。

テストスイートにユーザー定義のテストオーケストレーターが含まれていない場合は、IDT によってテストオーケストレーターが生成されます。

デフォルトのテストオーケストレーターには以下の機能があります。

- テストの実行者に、テストスイート全体ではなく、特定のテストグループを選択して実行する機能を提供する。
- 特定のテストグループが選択されていない場合、テストスイート内のすべてのテストグループをランダムな順序で実行する。
- レポートを生成し、各テストグループおよびテストケースのテスト結果を示すコンソールサマリーを出力する。

IDT テストオーケストレーターは、テストオーケストレーターに置き換えられます。テストスイートの開発には、IDT テストオーケストレーターではなくテストオーケストレータを使用することを強くお勧めします。テストオーケストレーターでは、以下の機能が改善されています。

- IDT ステートマシンが命令型を使用するのに対し、宣言型を使用します。これにより、どのテストを実行するか、およびいつそれを実行するかを指定できます。
- 特定のグループ処理、レポート生成、エラー処理、結果追跡を管理し、これらのアクションを手動管理する必要がないようにします。
- デフォルトでコメントをサポートする YAML 形式を使用します。
- 同じワークフローを定義するのに必要なディスク容量が、テストオーケストレーターより 80% 少なくなります。
- テスト前検証を追加し、誤ったテスト ID や依存関係の循環がワークフロー定義に含まれていないことを検証します。

テストオーケストレーター形式

次のテンプレートを使用して、独自の *<custom-test-suite-folder>*/suite/test_orchestrator.yaml ファイルを構成できます。

```
Aliases:
  string: context-expression

ConditionalTests:
  - Condition: context-expression
    Tests:
      - test-descriptor

Order:
  - - group-descriptor
    - group-descriptor

Features:
  - Name: feature-name
    Value: support-description
    Condition: context-expression
    Tests:
        - test-descriptor
    OneOfTests:
        - test-descriptor
    IsRequired: boolean
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Aliases`

オプション。コンテキスト式にマップするユーザー定義の文字列。エイリアスを使用すると、テストオーケストレーター設定でコンテキスト式を識別するためのフレンドリ名を生成できます。これは、複雑なコンテキスト式や、複数の場所で使用する式を作成する場合に特に便利です。

コンテキスト式を使用するとコンテキストクエリを保存でき、これにより他の IDT 設定からデータにアクセスできるようになります。詳細については、「コンテキスト内のデータにアクセスする」を参照してください。

Example 例

```
Aliases:
```

```
    FizzChosen: "'{{$pool.features[?(@.name == 'Fizz')].value[0]}}' == 'yes'"
    BuzzChosen: "'{{$pool.features[?(@.name == 'Buzz')].value[0]}}' == 'yes'"
    FizzBuzzChosen: "'{{$aliases.FizzChosen}}' && '{{$aliases.BuzzChosen}}'"
```

`ConditionalTests`

オプション。条件と、条件が満たされたときに実行される、対応するテストケースのリスト。各条件には複数のテストケースを割り当てることができますが、特定のテストケースを割り当てることができる条件は 1 つだけです。

デフォルトでは、IDT はこのリストの条件に割り当てられていないテストケースをすべて実行します。このセクションを指定しない場合、IDT はテストスイートのすべてのテストグループを実行します。

`ConditionalTests` リストの各項目には以下のパラメータが含まれます。

`Condition`

ブール値に評価されるコンテキスト式。評価値が true の場合、IDT は `Tests` パラメータで指定されたテストケースを実行します。

`Tests`

テスト記述子のリスト。

各テスト記述子は、テストグループ ID と 1 つ以上のテストケース ID を使用して、特定のテストグループから実行する個々のテストを識別します。テスト記述子は以下の形式を使用します。

```
GroupId: group-id
CaseIds: [test-id, test-id] # optional
```

Example 例

次の例は、`Aliases` として定義できる汎用コンテキスト式を使用します。

```
ConditionalTests:
    - Condition: "{{$aliases.Condition1}}"
      Tests:
          - GroupId: A
```

```
        - GroupId: B
    - Condition: "{{$aliases.Condition2}}"
      Tests:
        - GroupId: D
    - Condition: "{{$aliases.Condition1}} || {{$aliases.Condition2}}"
      Tests:
        - GroupId: C
```

定義された条件に基づき、IDT は次のようにテストグループを選択します。

- Condition1 が真の場合、IDT はテストグループ A、B、C のテストを実行します。
- Condition2 が真の場合、IDT はテストグループ C と D のテストを実行します。

`Order`

オプション。テストを実行する順序。テストの順序はテストグループレベルで指定します。このセクションを指定しない場合、IDT はすべての適用可能なテストグループをランダムな順序で実行します。`Order` の値は、グループ記述子リストのリストです。`Order` のリストに記載していないテストグループは、記載されている他のテストグループと並行して実行できます。

各グループ記述子リストには 1 つ以上のグループ記述子が含まれ、各記述子で指定されたグループを実行する順序を特定します。個別のグループ記述子を定義するには、以下の形式を使用できます。

- *`group-id`*—既存のテストグループのグループ ID。
- [*`group-id`*, *`group-id`*]—相互に任意の順序で実行できるテストグループのリスト。
- "*"—ワイルドカード これは、現在のグループ記述子リストにまだ指定されていないすべてのテストグループのリストに相当します。

`Order` の値は、次の要件も満たしている必要があります。

- グループ記述子で指定するテストグループ ID は、テストスイートに存在する必要があります。
- 各グループ記述子リストには、少なくとも 1 つのテストグループが含まれている必要があります。
- 各グループ記述子リストには一意のグループ ID を含める必要があります。個々のグループ記述子内でテストグループ ID を繰り返すことはできません。
- グループ記述子リストは、最大 1 つのワイルドカードグループ記述子を持つことができます。ワイルドカードグループ記述子は、リストの最初または最後の項目でなければなりません。

Example 例

テストグループ A、B、C、D、E を含むテストスイートについて、次の例のリストに、IDT が最初にテストグループ A を実行し、次にテストグループ B を実行し、次いで C、D、E を任意の順序で実行するよう指定するためのさまざまな方法を示します。

- ```
  Order:
      - - A
        - B
        - [C, D, E]
  ```
- ```
  Order:
      - - A
        - B
        - "*"
  ```
- ```
  Order:
      - - A
        - B

      - - B
        - C

      - - B
        - D

      - - B
        - E
  ```

`Features`

オプション。IDT に `awsiotdevicetester_report.xml` ファイルに追加させる製品機能のリスト。このセクションを指定しない場合、IDT はレポートに製品機能を追加しません。

製品機能とは、デバイスが満たしている可能性のある特定の基準に関するユーザー定義の情報です。例えば、MQTT 製品機能には、デバイスが MQTT メッセージを適切に公開することを指定できます。`awsiotdevicetester_report.xml` では、製品機能は指定されたテストが合格したかどうかに応じて、`supported`、`not-supported`、またはカスタムのユーザー定義値に設定されます。

`Features` リストの各項目は以下のパラメータで構成されます。
  ```

`Name`

機能の名前。

`Value`

オプション。`supported` の代わりにレポートで使用するカスタム値。この値を指定しない場合、テスト結果に基づいて、IDT により機能値が `supported` または `not-supported` に設定されます。同じ機能を異なる条件でテストする場合、`Features` リストにおけるその機能のインスタンスごとにカスタム値を使用できます。IDT は、サポート条件の機能値を連結します。詳細については、以下を参照してください。

`Condition`

ブール値に評価されるコンテキスト式。評価値が true の場合、IDT はテストスイートを実行後、その機能をテストレポートに追加します。評価値が false の場合、テストはレポートに含まれません。

`Tests`

オプション。テスト記述子のリスト。機能をサポートするには、このリストで指定されるテストにすべて合格する必要があります。

このリストの各テスト記述子は、テストグループ ID と 1 つ以上のテストケース ID を使用して、特定のテストグループから実行する個々のテストを識別します。テスト記述子は以下の形式を使用します。

```
GroupId: group-id
CaseIds: [test-id, test-id] # optional
```

`Features` リストの各機能について、`Tests` か `OneOfTests` のどちらかを指定する必要があります。

`OneOfTests`

オプション。テスト記述子のリスト。機能をサポートするには、このリストで指定されているテストのうち少なくとも 1 つに合格する必要があります。

このリストの各テスト記述子は、テストグループ ID と 1 つ以上のテストケース ID を使用して、特定のテストグループから実行する個々のテストを識別します。テスト記述子は以下の形式を使用します。

```
GroupId: group-id
```

```
CaseIds: [test-id, test-id] # optional
```

Features リストの各機能について、Tests か OneOfTests のどちらかを指定する必要があります。

IsRequired

機能がテストレポートに必要かどうかを定義するブール値。デフォルト値は false です。

Example

テストオーケストレーターコンテキスト

テストオーケストレーターコンテキストは、実行中のテストオーケストレーターに利用可能なデータが含まれている読み取り専用 JSON ドキュメントです。テストオーケストレーターコンテキストは、テストオーケストレーターからのみアクセス可能で、テストフローを決定する情報が含まれています。例えば、テストの実行者によって userdata.json ファイルに設定された情報を使用して、特定のテストを実行する必要があるかどうかを決定できます。

テストオーケストレーターコンテキストは次の形式を使用します。

```
{
    "pool": {
        <device-json-pool-element>
    },
    "userData": {
        <userdata-json-content>
    },
    "config": {
        <config-json-content>
    }
}
```

pool

テスト実行用に選択されたデバイスプールに関する情報。選択されたデバイスプールのこの情報は、device.json ファイルで定義された、対応する最上位レベルのデバイスプール配列要素から取得されます。

userData

userdata.json ファイル内の情報。

`config`

`config.json` ファイル内の情報。

コンテキストは、JSONPath 表記法を使用してクエリできます。ステート定義における JSonPath クエリの構文は {{*query*}} です。テストオーケストレーターコンテキストからデータにアクセスする場合、各値が文字列、数値、またはブール値として評価されることを確認してください。

JSONPath 表記を使用してコンテキストのデータにアクセスする方法の詳細については、IDT コンテキストを使用するを参照してください。

IDT ステートマシンを構成する

> **Important**
>
> IDT v4.5.1 以降、このステートマシンは非推奨です。新しいテストオーケストレーターを使用することを強くお勧めします。詳細については、「IDT テストオーケストレーターを設定する」を参照してください。

ステートマシンは、テストスイートの実行フローを制御するコンストラクトです。テストスイートの開始ステートを決定し、ユーザー定義のルールに基づいてステートの移行を管理し、終了ステートに達するまでステートの移行を継続します。

テストスイートにユーザー定義のステートマシンが含まれていない場合は、IDT によってステートマシンが生成されます。デフォルトのステートマシンには、次の機能があります。

- テストの実行者に、テストスイート全体ではなく、特定のテストグループを選択して実行する機能を提供する。
- 特定のテストグループが選択されていない場合、テストスイート内のすべてのテストグループをランダムな順序で実行する。
- レポートを生成し、各テストグループおよびテストケースのテスト結果を示すコンソールサマリーを出力する。

IDT テストスイートのステートマシンは、次の基準を満たす必要があります。

- 各ステートが、IDT が実行する各アクション (テストグループの実行、レポートファイルの生成など) に対応する。

- ステートが移行すると、そのステートに関連付けられたアクションを実行する。
- 各ステートが、次のステートの移行ルールを定義する。
- 終了ステートが Succeed または Fail である。

ステートマシンの形式

次のテンプレートを使用して、独自の *<custom-test-suite-folder>*/suite/state_machine.json ファイルを構成できます。

```
{
  "Comment": "<description>",
  "StartAt": "<state-name>",
  "States": {
    "<state-name>": {
      "Type": "<state-type>",
      // Additional state configuration
    }

    // Required states
    "Succeed": {
      "Type": "Succeed"
    },
    "Fail": {
      "Type": "Fail"
    }
  }
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Comment`

ステートマシンの説明。

`StartAt`

IDT がテストスイートの実行を開始するステートの名前。StartAt の値は、States オブジェクトにリストされているいずれかのステートに設定する必要があります。

`States`

ユーザー定義のステート名を有効な IDT ステートにマッピングするオブジェクト。各 States.*`state-name`* オブジェクトには、*`state-name`* にマッピングされた有効なステートの定義が含まれています。

`States` オブジェクトには、`Succeed` ステートおよび `Fail` ステートを含める必要があります。有効なステートについては、有効なステートとステートの定義を参照してください。

有効なステートとステートの定義

このセクションでは、IDT ステートマシンで使用可能なすべての有効なステートのステート定義について説明します。以下に示すステートの一部は、テストケースレベルでの設定をサポートしています。ただし、絶対に必要な場合を除き、テストケースレベルではなく、テストグループレベルでステート移行ルールを設定することをお勧めします。

ステートの定義

- RunTask
- 選択
- 並行
- AddProductFeatures
- レポート
- LogMessage
- SelectGroup
- 失敗
- 成功

RunTask

`RunTask` ステートは、テストスイートで定義されているテストグループからテストケースを実行します。

```
{
    "Type": "RunTask",
    "Next": "<state-name>",
    "TestGroup": "<group-id>",
```

```
    "TestCases": [
        "<test-id>"
    ],
    "ResultVar": "<result-name>"
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Next`

現在のステートのアクションを実行した後に移行するステートの名前。

`TestGroup`

オプション。実行するテストグループの ID。この値を指定しない場合、IDT はテストの実行者が選択するテストグループを実行します。

`TestCases`

オプション。`TestGroup` に指定されたグループのテストケース ID の配列。IDT は、`TestGroup` と `TestCases` の値に基づいて、次のようにテストの実行動作を決定します。

- `TestGroup` と `TestCases` 両方が指定されている場合、IDT はテストグループから指定されたテストケースを実行します。
- `TestCases` が指定され、`TestGroup` が指定されていない場合、IDT は指定されたテストケースを実行します。
- `TestGroup` が指定され、`TestCases` が指定されていない場合は、IDT は指定されたテストグループ内のすべてのテストケースを実行します。
- `TestGroup` も `TestCases` も指定されていない場合、IDT は、テストの実行者が IDT CLI から選択したテストグループからすべてのテストケースを実行します。テストの実行者がグループを選択できるようにするには、`state_machine.json` ファイルに `RunTask` ステートと `Choice`ステート両方を含める必要があります。これを行う方法の例については、ステートマシンの例: ユーザーが選択したテストグループを実行するを参照してください。

  テストの実行者向けの IDT CLI コマンドを有効にする方法については「the section called “IDT CLI コマンドを有効にする”」を参照してください。

`ResultVar`

テスト実行の結果によって設定するコンテキスト変数の名前。`TestGroup` の値を指定しなかった場合は、この値を指定しないでください。IDT は、以下に基づいて、`ResultVar` に定義された変数を `true` または `false` に設定します。

- 変数名の形式が *text*_*text*_passed の場合、この値は、最初のテストグループのすべてのテストが合格したか、スキップされたかに設定されます。
- それ以外の場合、この値は、すべてのテストグループのすべてのテストが合格したか、スキップされたかに設定されます。

通常、RunTask ステートは、個々のテストケース ID を指定せずにテストグループ ID を指定するために使用されます。この指定により、IDT は指定されたテストグループ内のすべてのテストケースを実行します。このステートで実行されるすべてのテストケースは、ランダムな順序で並行して実行されます。ただし、すべてのテストケースが実行に 1 つのデバイスを必要とし、単一のデバイスしか使用できない場合は、テストケースは順次実行されます。

エラー処理

指定されたテストグループ ID またはテストケース ID のいずれかが有効でない場合、このステートは RunTaskError 実行エラーを発行します。またこのステートは、実行エラーに遭遇すると、ステートマシンコンテキスト内の hasExecutionError 変数を true に設定します。

選択

Choice ステートでは、ユーザー定義の条件に基づいて、移行先の次のステートを動的に設定できます。

```
{
    "Type": "Choice",
    "Default": "<state-name>",
    "FallthroughOnError": true | false,
    "Choices": [
        {
            "Expression": "<expression>",
            "Next": "<state-name>"
        }
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

Default

Choices に定義されているいずれの式も true に評価されない場合に移行先になるデフォルトのステート。

`FallthroughOnError`

オプション。このステートが式評価エラーに遭遇したときの動作を指定します。評価結果がエラーになったときに式をスキップしたい場合は `true` に設定します。一致する式がない場合、ステートマシンは `Default` ステートに移行します。`FallthroughOnError` 値は、指定されない場合、デフォルトで `false` になります。

`Choices`

現在のステートのアクションを実行した後に移行するステートを決定する式とステートの配列。

`Choices.Expression`

ブール値に評価される式文字列。式が `true` と評価された場合、ステートマシンは `Choices.Next` に定義されているステートに移行します。式文字列は、ステートマシンコンテキストから値を取得し、オペレーションを実行してブール値に到達します。ステートマシンコンテキストへのアクセスについては、「ステートマシンコンテキスト」を参照してください。

`Choices.Next`

`Choices.Expression` で定義されている式が `true` に評価された場合の移行先のステート名。

エラー処理

以下に示すケースでは、`Choice` ステートでエラー処理が必要になることがあります。

- choice 式の一部の変数が、ステートマシンのコンテキストに存在しない。
- 式の結果がブール値ではない。
- JSON 検索の結果が、文字列、数値、またはブール値ではない。

このステートのエラー処理に `Catch` ブロックを使用することはできません。ステートマシンがエラーに遭遇したときに、その実行を停止するには、`FallthroughOnError` を `false` に設定する必要があります。ただし、`FallthroughOnError` は `true` に設定し、ユースケースに応じて、次のいずれかの操作を実行することをお勧めします。

- アクセスしている変数が一部のケースに存在しないと考えられる場合は、`Default` の値と追加の `Choices` ブロックを使用して次のステートを指定します。
- 使用している変数が必ず存在するものである場合は、`Default` ステートを `Fail` に設定します。

並行

Parallel ステートでは、新しいステートマシンを互いに並列に定義して実行できます。

```
{
    "Type": "Parallel",
    "Next": "<state-name>",
    "Branches": [
        <state-machine-definition>
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

Next

現在のステートのアクションを実行した後に移行するステートの名前。

Branches

実行するステートマシン定義の配列。各ステートマシン定義には、それぞれの StartAt、Succeed、および Fail ステートを含める必要があります。この配列内のステートマシン定義は、各自の定義外のステートを参照することはできません。

> **Note**
>
> 各ブランチステートマシンは同じステートマシンコンテキストを共有するため、あるブランチに変数を設定し、別のブランチからそれらの変数を読み込むと、予期しない動作が発生する可能性があります。

Parallel ステートは、すべてのブランチステートマシンを実行してから次のステートに移行します。デバイスを必要とする各ステートは、デバイスが利用可能になるまで実行を待ちます。複数のデバイスが利用可能な場合、このステートは並行して複数のグループからテストケースを実行します。十分な数のデバイスが利用できない場合、テストケースは順次実行されます。テストケースは、並列して実行される場合、ランダムな順序で実行されるため、同じテストグループからのテストの実行に異なるデバイスが使用されることがあります。

エラー処理

実行エラーを処理するには、ブランチステートマシンと親ステートマシンの両方が、Fail ステートに移行していることを確認します。

ブランチステートマシンは親ステートマシンに実行エラーを送信しないため、ブランチステートマシンの実行エラーを処理するために Catch ブロックを使用することはできません。代わりに、共有ステートマシンコンテキストの hasExecutionErrors 値を使用します。これを行う方法の例については、ステートマシンの例: 2 つのテストグループを並行して実行するを参照してください。

AddProductFeatures

AddProductFeatures ステートでは、IDT によって生成される awsiotdevicetester_report.xml ファイルに製品機能を追加できます。

製品機能とは、デバイスが満たしている可能性のある特定の基準に関するユーザー定義の情報です。例えば、MQTT 製品機能には、デバイスが MQTT メッセージを適切に公開することを指定できます。レポートでは、製品機能は指定されたテストが合格したかどうかに応じて、supported、not-supported、カスタム値に設定されます。

> **Note**
>
> AddProductFeatures ステートだけではレポートは生成されません。レポートを生成するには、このステートが Report ステートに移行する必要があります。

```
{
    "Type": "Parallel",
    "Next": "<state-name>",
    "Features": [
        {
            "Feature": "<feature-name>",
            "Groups": [
                "<group-id>"
            ],
            "OneOfGroups": [
                "<group-id>"
            ],
            "TestCases": [
                "<test-id>"
            ],
            "IsRequired": true | false,
            "ExecutionMethods": [
                "<execution-method>"
            ]
```

```
        }
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Next`

現在のステートのアクションを実行した後に移行するステートの名前。

`Features`

`awsiotdevicetester_report.xml` ファイルに表示される製品機能の配列。

`Feature`

機能の名前。

`FeatureValue`

オプション。`supported` の代わりにレポートで使用するカスタム値。この値を指定しない場合、テスト結果に基づいて、機能値は `supported` または `not-supported` に設定されます。

`FeatureValue` にカスタム値を使用する場合は、同じ機能を異なる条件でテストできます。IDT は、サポート条件の機能値を連結します。例えば、以下の抜粋は、`MyFeature` 機能と 2 つの異なる機能値を示しています。

```
...
{
    "Feature": "MyFeature",
    "FeatureValue": "first-feature-supported",
    "Groups": ["first-feature-group"]
},
{
    "Feature": "MyFeature",
    "FeatureValue": "second-feature-supported",
    "Groups": ["second-feature-group"]
},
...
```

両方のテストグループが合格した場合、機能値は `first-feature-supported, second-feature-supported` に設定されます。

`Groups`

オプション。テストグループ ID の配列。機能をサポートするには、指定された各テストグループ内のすべてのテストが合格である必要があります。

`OneOfGroups`

オプション。テストグループ ID の配列。機能をサポートするには、指定されたテストグループうち、少なくとも 1 つのグループに含まれるすべてのテストが合格である必要があります。

`TestCases`

オプション。テストケース ID の配列。この値を指定すると、次のことが適用されます。

- 機能をサポートするには、指定されたすべてのテストケースが合格である必要があります。
- `Groups` には、テストグループ ID を 1 つだけ含める必要があります。
- `OneOfGroups` は指定できません。

`IsRequired`

オプション。この機能をレポートでオプション機能としてマークするには、`false` に設定します。デフォルト値は `true` です。

`ExecutionMethods`

オプション。`device.json` ファイルに指定された `protocol` 値と一致する実行メソッドの配列。この値を指定した場合、この機能をレポートに含めるには、テストの実行者はこの配列の値の 1 つに一致する `protocol` 値を指定する必要があります。この値を指定しない場合、この機能は常にレポートに含まれます。

`AddProductFeatures` ステートを使用するには、`RunTask` ステートの `ResultVar` の値を以下のいずれかの値に指定する必要があります。

- 個々のテストケース ID を指定した場合は、`ResultVar` を*`group-id_test-id`*`_passed` に指定します。
- 個々のテストケース ID を指定しなかった場合は、`ResultVar` を*`group-id`*`_passed` に指定します。

`AddProductFeatures` ステートは、次の方法でテスト結果をチェックします。

- テストケース ID を指定しなかった場合は、各テストグループの結果は、ステートマシンコンテキスト内の *group-id*_passed 変数の値から決定されます。
- テストケース ID を指定した場合は、各テストの結果は、ステートマシンコンテキスト内の *group-id_test-id*_passed 変数の値から決定されます。

エラー処理

このステートで指定されたグループ ID が有効なグループ ID でない場合、このステートで AddProductFeaturesError 実行エラーが発生します。またこのステートは、実行エラーに遭遇すると、ステートマシンコンテキスト内の hasExecutionErrors 変数を true に設定します。

レポート

Report ステートでは、*suite-name*_Report.xml ファイルと awsiotdevicetester_report.xml ファイルが生成されます。またこのステートでは、レポートがコンソールにストリーミングされます。

```
{
    "Type": "Report",
    "Next": "<state-name>"
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

Next

現在のステートのアクションを実行した後に移行するステートの名前。

テストの実行者がテスト結果を確認できるように、テスト実行フローの終了ステートの前に Report ステートに移行する必要があります。通常、このステートの次のステートは Succeed です。

エラー処理

このステートは、レポート生成時に問題に遭遇した場合、ReportError 実行エラーを発行します。

LogMessage

LogMessage ステートでは、test_manager.log ファイルが生成され、ログメッセージがコンソールにストリーミングされます。

```
{
```

```
    "Type": "LogMessage",
    "Next": "<state-name>"
    "Level": "info | warn | error"
    "Message": "<message>"
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Next`

現在のステートのアクションを実行した後に移行するステートの名前。

`Level`

ログメッセージを作成するエラーレベル。有効でないレベルを指定すると、エラーメッセージが生成され、そのレベルは破棄されます。

`Message`

ログに記録するメッセージ。

SelectGroup

`SelectGroup` ステートでは、ステートマシンコンテキストを更新して選択されたグループを示します。このステートで設定した値は、後続のすべての `Choice` ステートによって使用されます。

```
{
    "Type": "SelectGroup",
    "Next": "<state-name>"
    "TestGroups": [
        <group-id>"
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Next`

現在のステートのアクションを実行した後に移行するステートの名前。

`TestGroups`

選択済みとしてマークされるテストグループの配列。この配列の各テストグループ ID について、*group-id*_selected 変数がコンテキストで `true` に設定されます。IDT は、指定されたグ

ループが存在するかどうかを検証しないため、有効なテストグループ ID を指定するようにしてください。

失敗

Fail ステートは、ステートマシンが正しく実行されなかったことを示します。これはステートマシンの終了ステートです。各ステートマシンの定義にこのステートを含める必要があります。

```
{
    "Type": "Fail"
}
```

成功

Succeed ステートは、ステートマシンが正しく実行されたことを示します。これはステートマシンの終了ステートです。各ステートマシンの定義にこのステートを含める必要があります。

```
{
    "Type": "Succeed"
}
```

ステートマシンコンテキスト

ステートマシンコンテキストは、実行中のステートマシンに利用可能なデータが含まれている読み取り専用 JSON ドキュメントです。ステートマシンコンテキストは、ステートマシンからのみアクセス可能で、テストフローを決定する情報が含まれています。例えば、テストの実行者によって userdata.json ファイルに設定された情報を使用して、特定のテストを実行する必要があるかどうかを決定できます。

ステートマシンコンテキストでは、次の形式が使用されます。

```
{
    "pool": {
        <device-json-pool-element>
    },
    "userData": {
        <userdata-json-content>
    },
    "config": {
```

```
        <config-json-content>
    },
    "suiteFailed": true | false,
    "specificTestGroups": [
        "<group-id>"
    ],
    "specificTestCases": [
        "<test-id>"
    ],
    "hasExecutionErrors": true
}
```

pool

テスト実行用に選択されたデバイスプールに関する情報。選択されたデバイスプールのこの情報は、device.json ファイルで定義された、対応する最上位レベルのデバイスプール配列要素から取得されます。

userData

userdata.json ファイル内の情報。

config

config.json ファイル内の情報。

suiteFailed

この値は、ステートマシンが起動すると false に設定されます。テストグループが RunTask ステートで失敗した場合、この値はステートマシン実行の残りの時間の間 true に設定されます。

specificTestGroups

テストの実行者がテストスイート全体ではなく特定のテストグループを選択して実行する場合に、このキーが作成され、特定のテストグループ ID のリストが格納されます。

specificTestCases

テストの実行者がテストスイート全体ではなく特定のテストケースを選択して実行する場合に、このキーが作成され、特定のテストケース ID のリストが格納されます。

hasExecutionErrors

ステートマシンの起動時には存在しません。いずれかのステートが実行エラーに遭遇した場合に、この変数が作成され、ステートマシンの実行の残りの時間の間 true に設定されます。

コンテキストは、JSONPath 表記法を使用してクエリできます。ステート定義における JSonPath クエリの構文は {{$.*query*}} です。JSONPath クエリは、一部のステートではプレースホルダー文字列として使用できます。IDT は、プレースホルダー文字列をコンテキストから評価された JSONPath クエリの値に置き換えます。プレースホルダーは、次の値に使用できます。

- `RunTask` ステートの `TestCases` 値。
- `Choice` ステートの `Expression` 値。

ステートマシンコンテキストからデータにアクセスする場合は、次の条件を満たしていることを確認します。

- JSON パスが `$.` で始まっている。
- 各値が、文字列、数値、またはブール値として評価される。

JSONPath 表記を使用してコンテキストのデータにアクセスする方法の詳細については、IDT コンテキストを使用するを参照してください。

実行エラー

実行エラーとは、ステートの実行時にステートマシンが遭遇する、ステートマシン定義内のエラーです。IDT は、各エラーに関する情報を `test_manager.log` ファイルに記録し、ログメッセージをコンソールにストリーミングします。

実行エラーは、次の方法を使用して処理できます。

- ステート定義内に `Catch`ブロック を追加する。
- ステートマシンコンテキストで `hasExecutionErrors` 値 の値を確認する。

Catch

`Catch` を使用するには、ステート定義に以下を追加します。

```
"Catch": [
    {
        "ErrorEquals": [
            "<error-type>"
        ]
        "Next": "<state-name>"
    }
```

```
]
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Catch.ErrorEquals`

キャッチするエラータイプの配列。実行エラーが指定された値のいずれかと一致する場合、ステートマシンは、`Catch.Next` に指定されているステートに移行します。生成されるエラーのタイプの詳細については、各ステート定義を参照してください。

`Catch.Next`

現在のステートが、`Catch.ErrorEquals` に指定されている値のいずれかと一致する実行エラーに遭遇した場合に、移行する次のステート。

キャッチブロックは、いずれかが一致するまで順番に処理されます。どのエラーもキャッチブロックに指定されているエラーと一致しない場合、ステートマシンは実行を継続します。実行エラーは誤ったステート定義によって発生するため、ステートが実行エラーに遭遇したときは Fail ステートに移行することをお勧めします。

hasExecutionError

一部のステートは、実行エラーに遭遇した場合、エラーを発行するだけでなく、ステートマシンコンテキストの `hasExecutionError` 値も `true` に設定します。この値を使用して、エラーがいつ発生したかを特定してから、`Choice` ステートを使用してステートマシンを `Fail` ステートに移行することができます。

この方法には次の特徴があります。

- ステートマシンは、`hasExecutionError` に値が割り当てられていると開始しません。またこの値は、特定のステートによって設定されるまで得られません。つまり、明示的に `FallthroughOnError` の値を `false` に設定することによって、実行エラーが発生していない場合に、この値にアクセスする `Choice` ステートがステートマシンを停止しないようにする必要があります。
- `hasExecutionError` は、一度 `true` に設定されると、false に設定されることも、コンテキストから削除されることもありません。つまり、この値は `true` に設定された初回のみ有効であり、以降のすべてのステートに対して意味のある値を提供しないことを意味します。
- `hasExecutionError` 値は `Parallel` ステート内のすべてのブランチステートマシンで共有されるため、アクセスされる順序によっては、予期せぬ結果が発生する可能性があります。

これらの特性から、代わりに Catch ブロックを使用できる場合は、この方法を使用することはお勧めしません。

ステートマシンの例

このセクションでは、ステートマシンの設定の例を紹介します。

例

- ステートマシンの例: 1 つのテストグループを実行する
- ステートマシンの例: ユーザーが選択したテストグループを実行する
- ステートマシンの例: 製品機能が含まれる 1 つのテストグループを実行する
- ステートマシンの例: 2 つのテストグループを並行して実行する

ステートマシンの例: 1 つのテストグループを実行する

このステートマシンの動作:

- ID GroupA のテストグループを実行します。このテストグループは、group.json ファイルのスイート内に存在している必要があります。
- 実行エラーをチェックし、エラーが見つかった場合は Fail に移行します。
- エラーがない場合には、レポートを生成し、Succeed に移行します。エラーがある場合は、Fail に移行します。

```
{
    "Comment": "Runs a single group and then generates a report.",
    "StartAt": "RunGroupA",
    "States": {
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "Report",
            "TestGroup": "GroupA",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
```

```
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

ステートマシンの例: ユーザーが選択したテストグループを実行する

このステートマシンの動作:

- テストの実行者が特定のテストグループを選択したかどうかをチェックします。テストの実行者がテストケースを選択するには、テストグループも選択する必要があるため、ステートマシンは特定のテストケースはチェックしません。
- テストグループが選択されている場合:
  - 選択されたテストグループ内のテストケースを実行します。そのために、ステートマシンは、RunTask ステートでは、テストグループまたはテストケースを明示的に指定しません。
  - すべてのテストを実行した後にレポートを生成し、終了します。
- テストグループが選択されていない場合:
  - テストグループ GroupA のテストを実行します。
  - レポートを生成して終了します。

```
{
```

```
    "Comment": "Runs specific groups if the test runner chose to do that, otherwise
 runs GroupA.",
    "StartAt": "SpecificGroupsCheck",
    "States": {
        "SpecificGroupsCheck": {
            "Type": "Choice",
            "Default": "RunGroupA",
            "FallthroughOnError": true,
            "Choices": [
                {
                    "Expression": "{{$.specificTestGroups[0]}} != ''",
                    "Next": "RunSpecificGroups"
                }
            ]
        },
        "RunSpecificGroups": {
            "Type": "RunTask",
            "Next": "Report",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "Report",
            "TestGroup": "GroupA",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
```

```
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

ステートマシンの例: 製品機能が含まれる 1 つのテストグループを実行する

このステートマシンの動作:

- テストグループ GroupA を実行します。
- 実行エラーをチェックし、エラーが見つかった場合は Fail に移行します。
- FeatureThatDependsOnGroupA 機能を awsiotdevicetester_report.xml ファイルに追加します。
  - GroupA が合格である場合、機能を supported に設定します。
  - レポートでこの機能をオプションとしてマークしません。
- エラーがない場合には、レポートを生成し、Succeed に移行します。エラーがある場合は、Fail に移行します。

```
{
    "Comment": "Runs GroupA and adds product features based on GroupA",
    "StartAt": "RunGroupA",
    "States": {
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "AddProductFeatures",
            "TestGroup": "GroupA",
            "ResultVar": "GroupA_passed",
            "Catch": [
```

```
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "AddProductFeatures": {
            "Type": "AddProductFeatures",
            "Next": "Report",
            "Features": [
                {
                    "Feature": "FeatureThatDependsOnGroupA",
                    "Groups": [
                        "GroupA"
                    ],
                    "IsRequired": true
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

ステートマシンの例: 2 つのテストグループを並行して実行する

このステートマシンの動作:

- GroupA および GroupB テストグループを並行して実行します。ブランチステートマシンの RunTask ステートによってコンテキストに格納された ResultVar 変数が AddProductFeatures ステートに利用可能になります。
- 実行エラーをチェックし、エラーが見つかった場合は Fail に移行します。このステートマシンは、Catch ブロックを使用しません。この方法では、ブランチステートマシンの実行エラーが検出されないためです。
- 合格したグループに基づいて、awsiotdevicetester_report.xml ファイルに機能を追加します。
  - GroupA が合格である場合、機能を supported に設定します。
  - レポートでこの機能をオプションとしてマークしません。
- エラーがない場合には、レポートを生成し、Succeed に移行します。エラーがある場合は、Fail に移行します。

デバイスプールに 2 つのデバイスが構成されている場合、GroupA と GroupB 両方を同時に実行できます。ただし、GroupA または GroupB のどちらかに複数のテストが含まれている場合は、両方のデバイスがそれらのテストに割り当てられることがあります。デバイスが 1 つだけ構成されている場合、テストグループは順次実行されます。

```
{
    "Comment": "Runs GroupA and GroupB in parallel",
    "StartAt": "RunGroupAAndB",
    "States": {
        "RunGroupAAndB": {
            "Type": "Parallel",
            "Next": "CheckForErrors",
            "Branches": [
                {
                    "Comment": "Run GroupA state machine",
                    "StartAt": "RunGroupA",
                    "States": {
                        "RunGroupA": {
                            "Type": "RunTask",
                            "Next": "Succeed",
                            "TestGroup": "GroupA",
                            "ResultVar": "GroupA_passed",
```

```
                        "Catch": [
                            {
                                "ErrorEquals": [
                                    "RunTaskError"
                                ],
                                "Next": "Fail"
                            }
                        ]
                    },
                    "Succeed": {
                        "Type": "Succeed"
                    },
                    "Fail": {
                        "Type": "Fail"
                    }
                }
            },
            {
                "Comment": "Run GroupB state machine",
                "StartAt": "RunGroupB",
                "States": {
                    "RunGroupA": {
                        "Type": "RunTask",
                        "Next": "Succeed",
                        "TestGroup": "GroupB",
                        "ResultVar": "GroupB_passed",
                        "Catch": [
                            {
                                "ErrorEquals": [
                                    "RunTaskError"
                                ],
                                "Next": "Fail"
                            }
                        ]
                    },
                    "Succeed": {
                        "Type": "Succeed"
                    },
                    "Fail": {
                        "Type": "Fail"
                    }
                }
            }
        ]
```

```
        },
        "CheckForErrors": {
            "Type": "Choice",
            "Default": "AddProductFeatures",
            "FallthroughOnError": true,
            "Choices": [
                {
                    "Expression": "{{$.hasExecutionErrors}} == true",
                    "Next": "Fail"
                }
            ]
        },
        "AddProductFeatures": {
            "Type": "AddProductFeatures",
            "Next": "Report",
            "Features": [
                {
                    "Feature": "FeatureThatDependsOnGroupA",
                    "Groups": [
                        "GroupA"
                    ],
                    "IsRequired": true
                },
                {
                    "Feature": "FeatureThatDependsOnGroupB",
                    "Groups": [
                        "GroupB"
                    ],
                    "IsRequired": true
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
```

```
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

IDT テストケース実行可能ファイルを作成する

テストケース実行可能ファイルは、次の方法でテストスイートフォルダ内に作成して配置できます。

- test.json ファイル内の引数または環境変数を使用して実行するテストを決定するテストスイートの場合は、テストスイート全体に対して 1 つのテストケース実行可能ファイルを作成することも、テストスイート内のテストグループごとに 1 つのテスト実行可能ファイルを作成することもできます。
- 指定したコマンドに基づいて特定のテストを実行するテストスイートの場合は、テストスイート内のテストケースごとに 1 つのテストケース実行可能ファイルを作成します。

テストを作成するユーザーは、ユースケースに適したアプローチを決定し、それに応じてテストケースの実行可能ファイルを設定できます。各 test.json ファイルに、正しいテストケース実行可能ファイルのパスを指定していること、および指定した実行可能ファイルが正常に実行されることを確認してください。

すべてのデバイスにテストケースを実行する準備が整うと、IDT は以下のファイルを読み取ります。

- test.json。選択されたテストケースが、開始するプロセスと設定する環境変数を決定するために使用します。
- suite.json。テストスイートが、設定する環境変数を決定するために使用します。

IDT は、test.json ファイルで指定されているコマンドと引数に基づいて、必要なテスト実行可能プロセスを開始し、必要な環境変数をプロセスに渡します。

IDT クライアント SDK を使用する

IDT クライアント SDK を使用すると、IDT とテスト対象のデバイスとのやり取りに使用できる API コマンドを使用して、テスト実行可能ファイルにテストロジックを簡単に記述できます。現在、IDT では次の SDK が用意されています。

- IDT クライアント SDK for Python
- IDT クライアント SDK for Go
- IDT Client SDK for Java

これらの SDK は、*<device-tester-extract-location>*/sdks フォルダにあります。新しいテストケース実行可能ファイルを作成するときは、使用する SDK をテストケース実行可能ファイルが含まれるフォルダにコピーし、コード内で SDK を参照する必要があります。このセクションでは、テストケースの実行可能ファイルで使用できる API コマンドについて簡単に説明します。

このセクションの内容

デバイスとのやり取り

次のコマンドを使用すると、デバイスとのやり取りや接続管理のための追加の関数を実装せずに、テスト対象デバイスと通信することができます。

`ExecuteOnDevice`

テストスイートが、SSH または Docker シェル接続をサポートするデバイス上で、シェルコマンドを実行できるようにします。

`CopyToDevice`

テストスイートが、IDT を実行するホストマシンから、SSH または Docker シェル接続をサポートするデバイス上の指定された場所にローカルファイルをコピーできるようにします。

`ReadFromDevice`

テストスイートが、UART 接続をサポートするデバイスのシリアルポートから読み取りできるようにします。

> **Note**
>
> IDT は、コンテキストからのデバイスアクセス情報を使用して確立されたデバイスへの直接接続を管理しないため、テストケース実行可能ファイルでは、デバイスとやり取り用のこれらの API コマンドを使用することをお勧めします。ただし、これらのコマンドがテストケー

スの要件を満たしていない場合は、IDT コンテキストからデバイスアクセス情報を取得し、この情報を使用してテストスイートからデバイスに直接接続できます。
直接接続するには、テスト対象デバイスとリソースデバイスそれぞれの `device.connectivity` フィールドと `resource.devices.connectivity` フィールドの情報を取得します。IDT コンテキスト使用の詳細については、IDT コンテキストを使用するを参照してください。

IDT とのやり取り

次のコマンドを使用すると、テストスイートが IDT と通信できるようになります。

`PollForNotifications`

テストスイートが IDT からの通知をチェックできるようにします。

`GetContextValue` および `GetContextString`

テストスイートが IDT コンテキストから値を取得できるようにします。詳細については、IDT コンテキストを使用する を参照してください。

`SendResult`

テストスイートがテストケースの結果を IDT にレポートできるようにします。このコマンドは、テストスイートの各テストケースの最後に呼び出す必要があります。

ホストとのやり取り

次のコマンドを使用すると、テストスイートがホストマシンと通信できるようになります。

`PollForNotifications`

テストスイートが IDT からの通知をチェックできるようにします。

`GetContextValue` および `GetContextString`

テストスイートが IDT コンテキストから値を取得できるようにします。詳細については、IDT コンテキストを使用する を参照してください。

`ExecuteOnHost`

テストスイートがローカルマシン上でコマンドを実行できるようにし、IDT がテストケース実行可能ファイルのライフサイクルを管理できるようにします。

IDT CLI コマンドを有効にする

run-suite コマンド IDT CLI には、テストの実行者がテスト実行をカスタマイズするためのいくつかのオプションがあります。テストの実行者がこれらのオプションを使用してカスタムテストスイートを実行できるようにするには、IDT CLI のサポートを実装します。サポートを実装しなくてもテストは実行できますが、一部の CLI オプションは正しく機能しません。理想的なカスタマーエクスペリエンスを提供するために、IDT CLI で run-suite コマンドの次の引数のサポートを実装することをお勧めします。

timeout-multiplier

テストの実行中にすべてのタイムアウトに適用される 1.0 より大きい値を指定します。

テストの実行者は、この引数を使用して、実行するテストケースのタイムアウトを増やすことができます。テストの実行者が run-suite コマンドにこの引数を指定すると、IDT はこの値を使用して IDT_TEST_TIMEOUT 環境変数の値を計算し、IDT コンテキストの config.timeoutMultiplier フィールドを設定します。この引数をサポートするには、以下の手順を実行する必要があります。

- test.json ファイルのタイムアウト値を直接使用する代わりに、IDT_TEST_TIMEOUT 環境変数を読み取り、正しく計算されたタイムアウト値を取得します。
- IDT コンテキストから config.timeoutMultiplier 値を取得し、長時間実行されるタイムアウトに適用します。

タイムアウトイベントによる早期終了の詳細については、終了動作を指定するを参照してください。

stop-on-first-failure

障害が発生した場合に、IDT がすべてのテスト実行を停止するように指定します。

テストの実行者がこの引数を run-suite コマンドを指定すると、IDT は障害が発生するとすぐにテストの実行を停止します。ただし、テストケースが並行して実行されている場合、この設定によって予期しない結果につながる可能性があります。このサポートを実装するには、テストロジックを使用して、IDT がこのイベントに遭遇した場合に、実行中のすべてのテストケースに対して、実行を停止し、一時リソースをクリーンアップし、テスト結果を IDT にレポートするように指示します。障害発生時の早期終了の詳細については、終了動作を指定するを参照してください。

group-id および test-id

IDT が選択されたテストグループまたはテストケースのみを実行するように指定します。

テストの実行者は、run-suite コマンドでこれらの引数を使用して、以下のテスト実行可能ファイルの動作を指定できます。

- 指定されたテストスイート内のすべてのテストグループを実行する。
- 指定されたテストグループ内から選択したテストを実行する。

これらの引数をサポートするには、テストスイート用のステートオーケストレーターに、自分のテストオーケストレーターの RunTask ステートおよび Choice ステートのセットが含まれている必要があります。カスタムステートマシンを使用しない場合は、デフォルトの IDT テストオーケストレーターに必要なステートが含まれているため、追加のアクションを行う必要はありません。ただし、カスタムテストオーケストレーターを使用している場合は、サンプルとして ステートマシンの例: ユーザーが選択したテストグループを実行する を使用して、自分のテストオーケストレーターに必要なステートを追加してください。

IDT CLI コマンドの詳細については、カスタムテストスイートのデバッグと実行を参照してください。

イベントログの書き込み

テストの実行中は、イベントログとエラーメッセージをコンソールに書き込むために stdout と stderr にデータを送信します。コンソールメッセージの形式の詳細については、コンソールメッセージの形式を参照してください。

IDT がテストスイートの実行を終了すると、この情報は *<devicetester-extract-location>*/results/*<execution-id>*/logs フォルダにある test_manager.log ファイルでも利用可能になります。

各テストケースは、テスト実行のログ (テスト対象デバイスのログを含む) を *<device-tester-extract-location>*/results/*execution-id*/logs フォルダにある *<group-id>*_*<test-id>* ファイルに書き込むように設定できます。これを行うには、testData.logFilePath クエリを使用して IDT コンテキストからログファイルへのパスを取得し、そのパスにファイルを作成し、必要なコンテンツをそのファイルに書き込みます。IDT は、実行中のテストケースに基づいてこのパスを自動的に更新します。テストケースのログファイルを作成しないことを選択すると、そのテストケースのファイルは生成されません。

また、必要に応じて *<device-tester-extract-location>*/logs フォルダに追加のログファイルを作成するようにテキスト実行可能ファイルをセットアップすることもできます。ファイルが上書きされないように、ログファイル名に一意のプレフィックスを指定することをお勧めします。

IDT に結果をレポートする

IDT は、テスト結果を awsiotdevicetester_report.xml ファイルと *suite-name*_report.xml ファイルに書き込みます。これらのレポートファイルは、*<device-tester-extract-location>*/results/*<execution-id>*/ にあります。両レポートとも、テストスイート実行の結果をキャプチャします。IDT がこれらのレポートに使用するスキーマの詳細については、IDT テストの結果とログを確認するを参照してください。

*suite-name*_report.xml ファイルのコンテンツを取得するには、SendResult コマンドを使用して、テスト実行が終了する前に、テスト結果を IDT にレポートする必要があります。IDT は、テスト結果を見つけられない場合、テストケースのエラーを発行します。次の Python の抜粋は、テスト結果を IDT に送信するコマンドを示しています。

```
request-variable = SendResultRequest(TestResult(result))
client.send_result(request-variable)
```

API を使用して結果をレポートしない場合、IDT はテストアーティファクトフォルダでテスト結果を検索します。このフォルダのパスは、IDT コンテキストの testData.testArtifactsPath フィールドに格納されています。このフォルダで、IDT は、アルファベット順にソートされた最初の XML ファイルをテスト結果として使用します。

テストロジックが JUnit XML 結果を生成する場合は、結果を解析してから API を使用して IDT に送信する代わりに、アーティファクトフォルダ内の XML ファイルにテスト結果を書き込んで、直接 IDT に提供することができます。

この方法を使用する場合は、テストロジックによってテスト結果が正確に要約されていること、および *suite-name*_report.xml ファイルと同じ形式で結果ファイルがフォーマットされていることを確認してください。IDT は、次の例外を除き、提供されたデータの検証を実行しません。

- IDT は testsuites タグのすべてのプロパティを無視します。代わりに、レポートされた他のテストグループ結果からタグのプロパティを計算します。
- testsuite 内に少なくとも 1 つの testsuites タグが存在する必要があります。

IDT はすべてのテストケースで同じアーティファクトフォルダを使用し、テスト実行の終了後、次のテスト実行までに結果ファイルを削除しないため、この方法を使用すると、IDT が正しくないファイルを読み取った場合に、誤ったレポートが行われる可能性もあります。IDT が適切な結果を読み取れるように、すべてのテストケースで生成される XML 結果ファイルに同じ名前を使用して、各テストケースの結果を上書きすることをお勧めします。テストスイートのレポート作成に複合的なアプロー

チ (一部のテストケースには XML 結果ファイルを使用し、他のテストケースには API を使用して結果を送信する) を使用することもできますが、このアプローチはお勧めしません。

終了動作を指定する

テキスト実行可能ファイルは、テストケースが障害やエラー結果をレポートした場合でも、常に終了コード 0 で終了するように設定します。ゼロ以外の終了コードは、テストケースが実行されなかったこと、またはテストケース実行可能ファイルが結果を IDT に通知できなかったことを示す場合にのみ使用します。IDT は、0 以外の終了コードを受信すると、テスト実行を妨げるエラーが発生したとしてテストケースをマークします。

IDT は、以下に示すイベントが発生すると、終了前にテストケースに実行の停止を要求 (または想定) することがあります。以下の情報を使用して、テストケースから以下の各イベントを検出するようにテストケース実行可能ファイルを設定します。

タイムアウト

テストケースが、`test.json` ファイルで指定されたタイムアウト値よりも長く実行されたときに発生します。テストの実行者が `timeout-multiplier` 引数を使用してタイムアウト乗数を指定すると、IDT はこの乗数を使用してタイムアウト値を計算します。

このイベントを検出するには、IDT_TEST_TIMEOUT 環境変数を使用します。テストの実行者がテストを起動すると、IDT は IDT_TEST_TIMEOUT 環境変数の値を計算されたタイムアウト値 (秒単位) に設定し、その変数をテストケース実行可能ファイルに渡します。この変数の値を読み取って適切なタイマーを設定します。

割り込み

テストの実行者が IDT に割り込むと発生します。例えば、Ctrl+C を押した時です。

ターミナルはすべての子プロセスに通知を伝播するため、割り込み通知を検出する通知ハンドラをテストケースに簡単に設定できます。

または、API を定期的にポーリングして、`PollForNotifications` API 応答の `CancellationRequested` ブール値をチェックできます。IDT は割り込み通知を受信すると、`CancellationRequested` ブールの値を `true` に設定します。

最初の失敗時に停止する

現在のテストケースと並行して実行中のテストケースが失敗し、テストの実行者が `stop-on-first-failure` 引数を使用して、障害の発生時に IDT が実行を停止するように設定しているときに発生します。

このイベントを検出するには、PollForNotifications API を定期的にポーリングして、API レスポンスの CancellationRequested ブールの値をチェックします。IDT は、最初の障害発生時に停止するように設定されている場合、障害に遭遇すると、CancellationRequested ブールの値を true に設定します。

これらのいずれかのイベントが発生すると、IDT は現在実行中のテストケースの実行が終了するまで 5 分間待機します。実行中のすべてのテストケースが 5 分以内に終了しない場合、IDT は各プロセスを強制的に停止させます。IDT は、プロセスの終了前にテスト結果を受け取らなかった場合、テストケースをタイムアウトしたとしてマークします。ベストプラクティスとして、いずれかのイベントが発生したときは、テストケースが以下のアクションを実行するようにします。

1. 通常のテストロジックの実行を停止する。
2. テスト対象デバイスのテストアーティファクトなど、すべての一時的なリソースをクリーンアップする。
3. テスト結果 (テストの失敗やエラーなど) を IDT にレポートする。
4. 終了する。

IDT コンテキストを使用する

IDT がテストスイートを実行するとき、テストスイートは、各テストの実行方法の決定に使用できる一連のデータにアクセスできます。このデータは IDT コンテキストと呼ばれます。例えば、テストの実行者によって userdata.json ファイルに提供されるユーザーデータ設定は、IDT コンテキスト内でテストスイートに提供されます。

IDT コンテキストは、読み取り専用の JSON ドキュメントと考えることができます。テストスイートは、オブジェクト、配列、数値などの標準 JSON データ型を使用して、コンテキストからデータを取得することや、コンテキストにデータを書き込むことができます。

コンテキストスキーマ

IDT コンテキストは次の形式を使用します。

```
{
    "config": {
        <config-json-content>
        "timeoutMultiplier": timeout-multiplier
    },
```

```
    "device": {
        <device-json-device-element>
    },
    "devicePool": {
        <device-json-pool-element>
    },
    "resource": {
        "devices": [
            {
                <resource-json-device-element>
                "name": "<resource-name>"
            }
        ]
    },
    "testData": {
        "awsCredentials": {
            "awsAccessKeyId": "<access-key-id>",
            "awsSecretAccessKey": "<secret-access-key>",
            "awsSessionToken": "<session-token>"
        },
        "logFilePath": "/path/to/log/file"
    },
    "userData": {
        <userdata-json-content>
    }
}
```

config

config.json ファイル からの情報。config フィールドには、次の追加フィールドも含まれます。

config.timeoutMultiplier

テストスイートによって使用される任意のタイムアウト値の乗数。この値は、IDT CLI からテストの実行者によって指定されます。デフォルト値は 1 です。

device

テスト実行用に選択されたデバイスに関する情報。この情報は、選択されたデバイスの device.json ファイル の devices 配列要素に相当します。

`devicePool`

テスト実行用に選択されたデバイスプールに関する情報。この情報は、選択されたデバイスプールの `device.json` ファイルに定義されている最上位レベルのデバイスプール配列要素に相当します。

`resource`

`resource.json` ファイルからのリソースデバイスに関する情報。

`resource.devices`

この情報は、`devices` ファイルに定義されている `resource.json` 配列に相当します。各 `devices` 要素には、以下の追加フィールドが含まれています。

`resource.device.name`

リソースデバイスの名前。この値は、`test.json` ファイルで `requiredResource.name` 値に設定されます。

`testData.awsCredentials`

AWS クラウドに接続するためにテストによって使用される AWS 認証情報。この情報は、`config.json` ファイルから取得されます。

`testData.logFilePath`

テストケースがログメッセージを書き込むログファイルへのパス。このファイルは、存在しない場合、テストスイートによって作成されます。

`userData`

テストの実行者によって [userdata.json ファイル](#) に提供された情報。

コンテキスト内のデータにアクセスする

コンテキストは、JSONPath 表記を使用して JSON ファイルからクエリすることも、`GetContextValue` および `GetContextString` API を使用してテキスト実行可能ファイルからクエリすることもできます。IDT コンテキストにアクセスするための JSONPath 文字列の構文は、次のように異なります。

- `suite.json` および `test.json` では、{{*query*}} を使用します。つまり、式を開始するためにルート要素 `$.` を使用しません。

- `test_orchestrator.yaml` では {{*query*}} を使用します。

  非推奨のステートマシンを使用する場合、`state_machine.json` では {{$.*query*}} を使用します。
- API コマンドでは、コマンドに応じて *query* または {{$.*query*}} を使用します。詳細については、SDK のインラインドキュメントを参照してください。

次の表に、一般的な JSONPath 式の演算子を示します。

Operator	Description
`$`	The root element. Because the top-level context value for IDT is an object, you will typically use `$.` to start your queries.
`.childName`	Accesses the child element with name `childName` from an object. If applied to an array, yields a new array with this operator applied to each element. The element name is case sensitive. For example, the query to access the `awsRegion` value in the `config` object is `$.config.awsRegion` .
`[start:end]`	Filters elements from an array, retrieving items beginning from the `start` index and going up to the `end` index, both inclusive.
`[index1, index2, ... , indexN]`	Filters elements from an array, retrieving items from only the specified indices.
`[?(expr)]`	Filters elements from an array using the `expr` expression. This expression must evaluate to a boolean value.

フィルター式を作成するには、次の構文を使用します。

```
<jsonpath> | <value> operator <jsonpath> | <value>
```

この構文の説明は次のとおりです。

- `jsonpath` は、標準 JSON 構文を使用する JSONPath です。
- `value` は、標準 JSON 構文を使用するカスタム値です。
- `operator` は、以下のいずれかの演算子です。
  - < (未満)
  - <= (以下)
  - == (等しい)

    式内の JSONPath または値が配列、ブール値、またはオブジェクト値である場合は、これがユーザーに使用可能な唯一の二項演算子です。
  - >= (以上)
  - > (次より大きい)
  - =~ (正規表現の一致)。この演算子をフィルター式で使用するには、式の左側の JSONPath または値が文字列に評価される必要があり、右側が RE2 構文 に従ったパターン値である必要があります。

{{*query*}} 形式の JSONPath クエリは、プレースホルダ文字列として、`test.json` ファイルの `args` および `environmentVariables` フィールド内と、`suite.json` ファイルの `environmentVariables` フィールド内で使用できます。IDT はコンテキスト検索を実行し、クエリの評価値をフィールドに入力します。例えば、`suite.json` ファイルでは、プレースホルダー文字列を使用して、各テストケースとともに変化する環境変数の値を指定できます。IDT は、環境変数に各テストケースの正しい値を入力します。ただし、`test.json` ファイルおよび `suite.json` ファイルでプレースホルダー文字列を使用する場合は、クエリに次の考慮事項が適用されます。

- クエリに含まれる各 `devicePool` キーは、すべて小文字にする必要があります。つまり、代わりに `devicepool` を使用します。
- 配列には、文字列の配列のみを使用できます。さらに、配列は非標準の `item1,item2,...,itemN` の形式を使用します。配列は、要素が 1 つしか含まれていない場合、`item` としてシリアル化され、文字列フィールドと区別がつかなくなります。
- プレースホルダーを使用してコンテキストからオブジェクトを取得することはできません。

これらの事項を考慮して、テストロジックのコンテキストへのアクセスには、`test.json` ファイルおよび `suite.json` ファイルのプレースホルダー文字列ではなく、可能な限り API を使用すること

をお勧めします。ただし、環境変数として設定する単一の文字列を取得するときは、JSONPath プレースホルダーを使用する方が便利な場合があります。

テストの実行者向けの設定の設定

カスタムテストスイートを実行するには、テストの実行者は、実行するテストスイートに基づいて設定を設定する必要があります。設定は、*<device-tester-extract-location>*/configs/ フォルダにある設定ファイルテンプレートに基づいて指定します。必要に応じて、テストの実行者は、IDT が AWS クラウドへの接続に使用する AWS 認証情報も設定する必要があります。

テストを作成するユーザーは、テストスイートをデバッグするために、以下に示すファイルの設定が必要になります。また、テストスイートを実行するために必要な以下の設定を設定できるように、テストの実行者に指示を提供する必要があります。

device.json の設定

device.json ファイルには、テストが実行されるデバイスに関する情報 (IP アドレス、ログイン情報、オペレーティングシステム、CPU アーキテクチャなど) が含まれています。

テストの実行者は、*<device-tester-extract-location>*/configs/ フォルダにある次のテンプレート device.json ファイルを使用してこの情報を指定できます。

```
[
    {
        "id": "<pool-id>",
        "sku": "<pool-sku>",
        "features": [
            {
                "name": "<feature-name>",
                "value": "<feature-value>",
                "configs": [
                    {
                        "name": "<config-name>",
                        "value": "<config-value>"
                    }
                ],
            }
        ],
        "devices": [
            {
                "id": "<device-id>",
```

```
                "connectivity": {
                    "protocol": "ssh | uart | docker",
                    // ssh
                    "ip": "<ip-address>",
                    "port": <port-number>,
                    "auth": {
                        "method": "pki | password",
                        "credentials": {
                            "user": "<user-name>",
                            // pki
                            "privKeyPath": "/path/to/private/key",

                            // password
                            "password": "<password>",
                        }
                    },

                    // uart
                    "serialPort": "<serial-port>",

                    // docker
                    "containerId": "<container-id>",
                    "containerUser": "<container-user-name>",
                }
            }
        ]
    }
]
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`id`

デバイスプールと呼ばれるデバイスのコレクションを一意に識別するユーザー定義の英数字の ID。プールに属するデバイスには、同一のハードウェアが必要です。テストスイートを実行する場合、プールのデバイスを使用してワークロードを並列化します。複数のデバイスを使用して異なるテストを実行します。

`sku`

テスト対象デバイスを一意に識別する英数字の値。SKU は、認定されたデバイスの追跡に使用されます。

> **Note**
>
> AWS Partner Device Catalog にボードを出品する場合は、ここで指定する SKU と出品プロセスで使用する SKU が一致しなければなりません。

`features`

オプションです。デバイスでサポートされている機能を含む配列。デバイス機能は、テストスイートに設定するユーザー定義の値です。テストの実行者に、`device.json` ファイルに含める機能名および値に関する情報を提供する必要があります。例えば、他のデバイスの MQTT サーバーとして機能するデバイスをテストする場合は、`MQTT_QOS` という名前の機能に対する特定のサポートレベルを検証するようにテストロジックを設定します。テストの実行者は、この機能名を指定し、デバイスによってサポートされる QOS レベルにその機能値を設定します。指定された情報は、`devicePool.features` クエリを使用して [IDT context] (IDT コンテキスト) から、または `pool.features` クエリを使用して[test orchestrator context] (テストオーケストレーターコンテキスト) から取得できます。

`features.name`

機能の名前。

`features.value`

サポートされている機能値。

`features.configs`

機能の構成設定 (必要な場合)。

`features.config.name`

構成設定の名前。

`features.config.value`

サポートされている設定値。

`devices`

テスト対象のプール内のデバイスの配列。少なくとも 1 つのデバイスが必要です。

`devices.id`

テスト対象のデバイスのユーザー定義の一意の識別子。

`connectivity.protocol`

このデバイスと通信するために使用される通信プロトコル。プール内の各デバイスは、同じプロトコルを使用する必要があります。

現在、サポートされている値は、物理デバイス用の `ssh` および `uart` と、Docker コンテナ用の `docker` のみです。

`connectivity.ip`

テスト対象のデバイスの IP アドレス。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.port`

オプションです。SSH 接続に使用するポート番号。

デフォルト値は 22 です。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.auth`

接続の認証情報。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.auth.method`

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- `pki`
- `password`

`connectivity.auth.credentials`

認証に使用される認証情報。

`connectivity.auth.credentials.password`

テスト中のデバイスにサインインするためのパスワード。

この値は、`connectivity.auth.method` が `password` に設定されている場合にのみ適用されます。

`connectivity.auth.credentials.privKeyPath`

テスト中のデバイスにサインインするためのプライベートキーへの完全パス。

この値は、`connectivity.auth.method` が `pki` に設定されている場合にのみ適用されます。

`connectivity.auth.credentials.user`

テスト対象デバイスにサインインするためのユーザー名。

`connectivity.serialPort`

オプションです。デバイスが接続されているシリアルポート。

このプロパティは、`connectivity.protocol` が `uart` に設定されている場合にのみ適用されます。

`connectivity.containerId`

テスト対象の Docker コンテナのコンテナ ID または名前。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.containerUser`

オプションです。コンテナ内のユーザー名。デフォルト値は Dockerfile で指定されたユーザーです。

デフォルト値は 22 です。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

> **Note**
>
> テストの実行者がテストに対して誤ったデバイス接続を設定しているかどうかを確認するには、テストオーケストレーターコンテキストから

pool.Devices[0].Connectivity.Protocol を取得し、この値を Choice ステート内の予想値と比較します。正しくないプロトコルが使用されている場合は、LogMessage ステートを使用してメッセージを出力し、Fail ステートに移行します。
または、エラー処理コードを使用して、誤ったデバイスタイプによるテスト失敗をレポートすることもできます。

(オプション) userdata.json の設定

userdata.json ファイルには、device.json ファイルには指定されていない、テストスイートに必要とされる追加情報が含まれています。このファイルの形式は、テストスイートに定義されている userdata_scheme.json ファイルによって異なります。テストを作成するユーザーは、作成したテストスイートを実行するユーザーにこの情報を提供してください。

(オプション) resource.json の設定

resource.json ファイルには、リソースデバイスとして使用されるすべてのデバイスに関する情報が含まれています。リソースデバイスは、テスト対象のデバイスの特定の機能をテストするために必要なデバイスです。例えば、デバイスの Bluetooth 機能をテストするには、リソースデバイスを使用して、デバイスがリソースデバイスに正常に接続できるかどうかをテストできます。リソースデバイスはオプションで、必要な数だけリソースデバイスを要求できます。テストを作成するユーザーは、test.json ファイル を使用して、テストに必要なリソースデバイスの機能を定義します。テストの実行者は、resource.json ファイルを使用して、必要な機能を持つリソースデバイスのプールを指定します。作成したテストスイートを実行するユーザーに、以下の情報を提供してください。

テストの実行者は、*<device-tester-extract-location>*/configs/ フォルダにある次のテンプレート resource.json ファイルを使用してこの情報を指定できます。

```
[
    {
        "id": "<pool-id>",
        "features": [
            {
                "name": "<feature-name>",
                "version": "<feature-version>",
                "jobSlots": <job-slots>
            }
        ],
        "devices": [
            {
```

```
                "id": "<device-id>",
                "connectivity": {
                    "protocol": "ssh | uart | docker",
                    // ssh
                    "ip": "<ip-address>",
                    "port": <port-number>,
                    "auth": {
                        "method": "pki | password",
                        "credentials": {
                            "user": "<user-name>",
                            // pki
                            "privKeyPath": "/path/to/private/key",

                            // password
                            "password": "<password>",
                        }
                    },

                    // uart
                    "serialPort": "<serial-port>",

                    // docker
                    "containerId": "<container-id>",
                    "containerUser": "<container-user-name>",
                }
            }
        ]
    }
]
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`id`

デバイスプールと呼ばれるデバイスのコレクションを一意に識別するユーザー定義の英数字の ID。プールに属するデバイスには、同一のハードウェアが必要です。テストスイートを実行する場合、プールのデバイスを使用してワークロードを並列化します。複数のデバイスを使用して異なるテストを実行します。

`features`

オプションです。デバイスでサポートされている機能を含む配列。このフィールドに必要な情報は、テストスイートの test.json ファイル に定義されています。この情報によって、実行するテ

ストと、テストの実行方法が決まります。テストスイートに機能が必要ない場合は、このフィールドは必須ではありません。

`features.name`

機能の名前。

`features.version`

機能バージョン。

`features.jobSlots`

デバイスを同時に使用できるテストの数を示すための設定。デフォルト値は、1です。

`devices`

テスト対象のプール内のデバイスの配列。少なくとも 1 つのデバイスが必要です。

`devices.id`

テスト対象のデバイスのユーザー定義の一意の識別子。

`connectivity.protocol`

このデバイスと通信するために使用される通信プロトコル。プール内の各デバイスは、同じプロトコルを使用する必要があります。

現在、サポートされている値は、物理デバイス用の `ssh` および `uart` と、Docker コンテナ用の `docker` のみです。

`connectivity.ip`

テスト対象のデバイスの IP アドレス。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.port`

オプションです。SSH 接続に使用するポート番号。

デフォルト値は 22 です。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.auth`

接続の認証情報。

このプロパティは、connectivity.protocol が ssh に設定されている場合にのみ適用されます。

connectivity.auth.method

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- pki
- password

connectivity.auth.credentials

認証に使用される認証情報。

connectivity.auth.credentials.password

テスト中のデバイスにサインインするためのパスワード。

この値は、connectivity.auth.method が password に設定されている場合にのみ適用されます。

connectivity.auth.credentials.privKeyPath

テスト中のデバイスにサインインするためのプライベートキーへの完全パス。

この値は、connectivity.auth.method が pki に設定されている場合にのみ適用されます。

connectivity.auth.credentials.user

テスト対象デバイスにサインインするためのユーザー名。

connectivity.serialPort

オプションです。デバイスが接続されているシリアルポート。

このプロパティは、connectivity.protocol が uart に設定されている場合にのみ適用されます。

connectivity.containerId

テスト対象の Docker コンテナのコンテナ ID または名前。

このプロパティは、connectivity.protocol が ssh に設定されている場合にのみ適用されます。

`connectivity.containerUser`

オプションです。コンテナ内のユーザー名。デフォルト値は Dockerfile で指定されたユーザーです。

デフォルト値は 22 です。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

(オプション) config.json の設定

`config.json` ファイルには、IDT 向けの設定情報が含まれています。通常、テストの実行者は、IDT 用の AWS ユーザー認証情報、AWS リージョン (オプション) を指定することを除き、このファイルを変更する必要はありません。必要なアクセス許可が付与される AWS 認証情報が指定されると、AWS IoT Device Tester は使用状況メトリクスを収集して AWS に送信します。これはオプトイン機能で、IDT 機能を改善するために使用されます。詳細については、IDT 使用状況メトリクスを参照してください。

テストの実行者は、以下のいずれかの方法で AWS 認証情報を入手します。

- 認証情報ファイル

  IDT では、AWS CLI と同じ認証情報ファイルが使用されます。詳細については、「設定ファイルと認証情報ファイル」を参照してください。

  認証情報ファイルの場所は、使用しているオペレーティングシステムによって異なります。

  - macOS、Linux: `~/.aws/credentials`
  - Windows: `C:\Users\UserName\.aws\credentials`
- 環境変数

  環境変数は、オペレーティングシステムによって維持され、システムコマンドによって使用される変数です。SSH セッション中に定義された変数は、そのセッションの終了後は使用できません。IDT は、環境変数の `AWS_ACCESS_KEY_ID` と `AWS_SECRET_ACCESS_KEY` を使用して AWS 認証情報を保存します。

  これらの変数を Linux、macOS、または Unix で設定するには、export を使用します。

```
export AWS_ACCESS_KEY_ID=<your_access_key_id>
```

```
export AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

Windows でこれらの変数を設定するには、set を使用します。

```
set AWS_ACCESS_KEY_ID=<your_access_key_id>
set AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

IDT 用の AWS 認証情報を設定するには、テストの実行者は、auth フォルダにある config.json ファイルの *<device-tester-extract-location>*/configs/ セクションを編集します。

```
{
    "log": {
        "location": "logs"
    },
    "configFiles": {
        "root": "configs",
        "device": "configs/device.json"
    },
    "testPath": "tests",
    "reportPath": "results",
    "awsRegion": "<region>",
    "auth": {
        "method": "file | environment",
        "credentials": {
            "profile": "<profile-name>"
        }
    }
}
]
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

> **Note**
>
> このファイル内のすべてのパスは*##device-tester-extract-location#* を基準にして定義されます。

`log.location`

#device-tester-extract-location# のログフォルダへのパス。

`configFiles.root`

設定ファイルが含まれるフォルダへのパス。

`configFiles.device`

`device.json` ファイルへのパス。

`testPath`

テストスイートが含まれるフォルダへのパス。

`reportPath`

IDT がテストスイートを実行した後にテスト結果が含まれるフォルダへのパス。

`awsRegion`

オプションです。テストスイートが使用する AWS リージョン。設定されない場合、テストスイートは各テストスイートに指定されているデフォルトのリージョンを使用します。

`auth.method`

IDT が AWS 認証情報の取得に使用する方法。サポートされる値は、`file` (認証情報ファイルから認証情報を取得) と `environment` (環境変数を使用して認証情報を取得) です。

`auth.credentials.profile`

認証情報ファイルから使用する認証情報プロファイル。このプロパティは、`auth.method` が `file` に設定されている場合にのみ適用されます。

カスタムテストスイートのデバッグと実行

必要な設定 を終了すると、IDT はテストスイートを実行することができます。完全なテストスイートの実行時間は、ハードウェアとテストスイートの設定によって異なります。参照として、Raspberry Pi 3B に完全な AWS IoT Greengrass 適合性確認テストスイートを完了するために約 30 分かかります。

テストスイートの作成中に、IDT を使用してテストスイートをデバッグモードで実行すると、テストスイートを実行する前やテストの実行者に提供する前に、コードをチェックすることができます。

IDT をデバッグモードで実行する

テストスイートは、IDT に依存してデバイスとやり取りし、コンテキストを提供し、結果を受け取るため、IDT と通信しないと、IDE でテストスイートを簡単にデバッグすることはできません。そのために、IDT CLI は IDT をデバッグモードで実行できるようにする `debug-test-suite` コマンドを提供します。`debug-test-suite` で使用可能なオプションを表示するには、次のコマンドを実行します。

```
devicetester_[linux | mac | win_x86-64] debug-test-suite -h
```

IDT をデバッグモードで実行するとき、IDT は実際にテストスイートを起動したり、テストオーケストレーターを実行したりしません。代わりに、IDE と対話して IDE で実行されているテストスイートによるリクエストに応答して、コンソールにログを出力します。IDT はタイムアウトせず、手動で中断されるまで待機してから終了します。デバッグモードでは、IDT はテストオーケストレーターも実行せず、レポートファイルも生成しません。テストスイートをデバッグするには、通常 IDT が設定 JSON ファイルから取得する情報を、IDE を使用して提供する必要があります。以下の情報を提供してください。

- 各テストの環境変数と引数。IDT はこの情報を `test.json` または `suite.json` から読み込みません。
- リソースデバイスを選択するための引数。IDT はこの情報を `test.json` から読み込みません。

テストスイートをデバッグするには、次の手順を実行します。

1. テストスイートの実行に必要な構成設定ファイルを作成します。例えば、テストスイートが `device.json`、`resource.json`、および `user data.json` を必要とする場合は、必要に応じてこれらすべてを設定してください。
2. 次のコマンドを実行して IDT をデバッグモードにし、テストの実行に必要なデバイスを選択します。

   ```
   devicetester_[linux | mac | win_x86-64] debug-test-suite [options]
   ```

   このコマンドを実行すると、IDT はテストスイートからのリクエストを待機し、それらのリクエストに応答します。IDT は、IDT クライアント SDK がケースを処理するために必要な環境変数も生成します。
3. IDE で、`run` または `debug` 設定を使用して次の手順を実行します。

a. IDT で生成された環境変数の値を設定します。
b. `test.json` ファイルと `suite.json` ファイルに指定したすべての環境変数または引数の値を設定します。
c. 必要に応じてブレークポイントを設定します。

4. IDE でテストスイートを実行します。

   テストスイートは、必要に応じて何度でもデバッグして再実行できます。IDT はデバッグモードではタイムアウトしません。

5. デバッグが完了したら、IDT を中断してデバッグモードを終了します。

テストを実行する IDT CLI コマンド

次のセクションでは、IDT CLI コマンドについて説明します。

IDT v4.0.0

`help`

指定されたコマンドに関する情報を一覧表示します。

`list-groups`

特定のテストスイート内のグループを一覧表示します。

`list-suites`

使用可能なテストスイートを一覧表示します。

`list-supported-products`

IDT バージョン (この場合は AWS IoT Greengrass バージョン) のサポート対象製品と、現在の IDT バージョンで利用可能な AWS IoT Greengrass 適合性確認テストスイートのバージョンを一覧表示します。

`list-test-cases`

指定したテストグループのテストケースを一覧表示します。次のオプションがサポートされています。

- `group-id`。検索するテストグループ。このオプションは必須で、1 つのグループを指定する必要があります。

`run-suite`

デバイスプールに対してテストスイートを実行します。以下に、一般的に使用されるオプションの一部を示します。

- `suite-id`。実行するテストスイートのバージョン。指定しない場合、IDT は `tests` フォルダにある最新バージョンを使用します。
- `group-id`。実行するテストグループ (カンマ区切りリストとして)。指定しない場合、IDT はテストスイートのすべてのテストグループを実行します。
- `test-id`。実行するテストケース (カンマ区切りリストとして)。指定した場合は、`group-id` は 1 つのグループを指定する必要があります。
- `pool-id`。テストするデバイスプール。`device.json` ファイルに複数のデバイスプールが定義されている場合、テストの実行者は 1 つのプールを指定する必要があります。
- `timeout-multiplier`。テスト用の `test.json` ファイルに指定されているテスト実行タイムアウトを、ユーザー定義乗数を使用して変更するように IDT を設定します。
- `stop-on-first-failure`。最初に障害が発生した時点で実行を停止するように IDT を設定します。指定されたテストグループをデバッグするには、このオプションを `group-id` とともに使用する必要があります。
- `userdata`。テストスイートの実行に必要なユーザーデータ情報を含むファイルを設定します。テストスイートの `suite.json` ファイルで、`userdataRequired` が true に設定されている場合にのみ必要です。

`run-suite` オプションの詳細については、次の `help` オプションを使用してください。

```
devicetester_[linux | mac | win_x86-64] run-suite -h
```

`debug-test-suite`

デバッグモードでテストスイートを実行します。詳細については、「IDT をデバッグモードで実行する」を参照してください。

IDT テストの結果とログを確認する

このセクションでは、IDT がコンソールログとテストレポートを生成する形式について説明します。

コンソールメッセージの形式

AWS IoT Device Tester は、テストスイートを起動するときに、標準形式を使用してコンソールにメッセージを出力します。以下の抜粋は、IDT によって生成されるコンソールメッセージの例を示しています。

```
time="2000-01-02T03:04:05-07:00" level=info msg=Using suite: MyTestSuite_1.0.0
executionId=9a52f362-1227-11eb-86c9-8c8590419f30
```

コンソールメッセージの大半は、次のフィールドで構成されます。

`time`

ログに記録されたイベントの完全な ISO 8601 タイムスタンプ。

`level`

ログに記録されたイベントのメッセージレベル。通常、ログに記録されるメッセージレベルは、`info`、`warn`、または `error` のいずれかです。IDT は、早期終了の原因となる予期されるイベントが発生した場合は、`fatal` または `panic` メッセージを発行します。

`msg`

ログに記録されたメッセージ。

`executionId`

現在の IDT プロセスの一意の ID 文字列。この ID は、個々の IDT 実行を区別するために使用されます。

テストスイートから生成されたコンソールメッセージは、テスト対象のデバイスとテストスイート、テストグループ、IDT が実行するテストケースに関する追加情報を提供します。次の抜粋は、テストスイートから生成されたコンソールメッセージの例を示しています。

```
time="2000-01-02T03:04:05-07:00" level=info msg=Hello world! suiteId=MyTestSuite
groupId=myTestGroup testCaseId=myTestCase deviceId=my-device
executionId=9a52f362-1227-11eb-86c9-8c8590419f30
```

コンソールメッセージのテストスイート固有の部分には、次のフィールドが含まれています。

`suiteId`

現在実行中のテストスイートの名前。

`groupId`

現在実行中のテストグループの ID。

`testCaseId`

現在実行中のテストケースの ID。

`deviceId`

現在のテストケースが使用しているテスト対象デバイスの ID。

IDT のテスト実行完了時にテストサマリーをコンソールに出力するには、テストオーケストレーターに Report ステートを含める必要があります。テストサマリーには、テストスイート、実行された各グループのテスト結果、生成されたログファイルとレポートファイルの場所に関する情報が含まれています。次の例は、テストサマリーメッセージを示しています。

```
========== Test Summary ==========
Execution Time:     5m00s
Tests Completed:    4
Tests Passed:       3
Tests Failed:       1
Tests Skipped:      0
----------------------------------
Test Groups:
    GroupA:         PASSED
    GroupB:         FAILED
----------------------------------
Failed Tests:
    Group Name: GroupB
        Test Name: TestB1
            Reason: Something bad happened
----------------------------------
Path to IoT Device Tester Report: /path/to/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/logs
Path to Aggregated JUnit Report: /path/to/MyTestSuite_Report.xml
```

AWS IoT Device Tester のレポートスキーマ

`awsiotdevicetester_report.xml` は、次の情報が含まれる署名済みレポートです。

- IDT バージョン。

- テストスイートのバージョン。
- レポートの署名に使用されるレポートの署名とキー。
- device.json ファイルで指定されているデバイス SKU とデバイスプール。
- テストされた製品のバージョンとデバイスの機能。
- テスト結果の概要の集計。この情報は、*suite-name*_report.xml ファイルに含まれている情報と同じです。

```
<apnreport>
    <awsiotdevicetesterversion>idt-version</awsiotdevicetesterversion>
    <testsuiteversion>test-suite-version</testsuiteversion>
    <signature>signature</signature>
    <keyname>keyname</keyname>
    <session>
        <testsession>execution-id</testsession>
        <starttime>start-time</starttime>
        <endtime>end-time</endtime>
    </session>
    <awsproduct>
        <name>product-name</name>
        <version>product-version</version>
        <features>
            <feature name="<feature-name>" value="supported | not-supported | <feature-
value>" type="optional | required"/>
        </features>
    </awsproduct>
    <device>
        <sku>device-sku</sku>
        <name>device-name</name>
        <features>
            <feature name="<feature-name>" value="<feature-value>"/>
        </features>
        <executionMethod>ssh | uart | docker</executionMethod>
    </device>
    <devenvironment>
        <os name="<os-name>"/>
    </devenvironment>
    <report>
        <suite-name-report-contents>
    </report>
</apnreport>
```

awsiotdevicetester_report.xml ファイルには、テスト対象の製品および一連のテストの実行後に検証された製品機能に関する情報を含む <awsproduct> タグが含まれています。

<awsproduct> タグで使用される属性

name

テスト対象の製品の名前。

version

テスト対象の製品のバージョン。

features

検証された機能です。required としてマークされている機能は、テストスイートがデバイスを検証するために必要です。次のスニペットは、この情報が awsiotdevicetester_report.xml ファイルで表示される方法を示します。

```
<feature name="ssh" value="supported" type="required"></feature>
```

optional としてマークされている機能は、検証に必須ではありません。次のスニペットは、オプションの機能を示しています。

```
<feature name="hsi" value="supported" type="optional"></feature>

<feature name="mqtt" value="not-supported" type="optional"></feature>
```

テストスイートのレポートスキーマ

*suite-name*_Result.xml レポートは JUnit XML 形式です。Jenkins、Bamboo などのように継続的な統合 (CI) と継続的なデプロイ (CD) のプラットフォームに統合することができます。このレポートには、テスト結果の概要の集計が含まれています。

```
<testsuites name="<suite-name> results" time="<run-duration>" tests="<number-of-test>"
 failures="<number-of-tests>" skipped="<number-of-tests>" errors="<number-of-tests>"
 disabled="0">
    <testsuite name="<test-group-id>" package="" tests="<number-of-tests>"
 failures="<number-of-tests>" skipped="<number-of-tests>" errors="<number-of-tests>"
 disabled="0">
        <!--success-->
```

```
        <testcase classname="<classname>" name="<name>" time="<run-duration>"/>
        <!--failure-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <failure type="<failure-type>">
                reason
            </failure>
        </testcase>
        <!--skipped-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <skipped>
                reason
            </skipped>
        </testcase>
        <!--error-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <error>
                reason
            </error>
        </testcase>
    </testsuite>
</testsuites>
```

awsiotdevicetester_report.xml と *suite-name*_report.xml 両方のレポートセクションには、実行されたテストとその結果が一覧表示されます。

最初の XML タグ <testsuites> には、テストの実行の概要が含まれています。例:

```
<testsuites name="MyTestSuite results" time="2299" tests="28" failures="0" errors="0"
 disabled="0">
```

<testsuites> タグで使用される属性

name

テストスイートの名前。

time

スイートの実行所要時間 (秒)。

tests

実行されたテストの数。

`failures`

実行されたテストのうち、合格しなかったものの数。

`errors`

IDT で実行できなかったテストの数。

`disabled`

この属性は使用されていないため無視できます。

テストに障害やエラーが発生した場合は、<testsuites> XML タグを確認することで、障害の生じたテストを特定できます。<testsuites> タグ内の <testsuite> XML タグは、テストグループのテスト結果の要約を示します。例:

```
<testsuite name="combination" package="" tests="1" failures="0" time="161" disabled="0"
 errors="0" skipped="0">
```

形式は <testsuites> タグと似ていますが、使用されていないため無視できる skipped という属性があります。各 <testsuite> XML タグ内には、テストグループの実行されたテスト別の <testcase> タグがあります。例:

```
<testcase classname="Security Test" name="IP Change Tests" attempts="1"></testcase>>
```

<testcase> タグで使用される属性

`name`

テストの名前。

`attempts`

IDT でテストケースを実行した回数。

テストに障害やエラーが発生した場合、<failure> タグまたは <error> タグがトラブルシューティングのための情報とともに <testcase> タグに追加されます。例:

```
<testcase classname="mcu.Full_MQTT" name="MQTT_TestCase" attempts="1">
 <failure type="Failure">Reason for the test failure</failure>
 <error>Reason for the test execution error</error>
</testcase>
```

IDT 使用状況メトリクス

必要なアクセス許可が付与される AWS 認証情報を指定すると、AWS IoT Device Tester は使用状況メトリクスを収集して AWS に送信します。これはオプトイン機能で、IDT 機能を改善するために使用されます。IDT は次のような情報を収集します。

- IDT の実行に使用される AWS アカウント ID
- テストの実行に使用される IDT AWS CLI コマンド
- 実行されるテストスイート
- *<device-tester-extract-location>* フォルダにあるテストスイート
- デバイスプール内に設定されているデバイスの数
- テストケース名と実行時間
- テストに合格したか、失敗したか、エラーが発生したか、スキップされたかなどのテスト結果情報
- テストされた製品の機能
- 予期せぬ終了、早期終了などの IDT 終了動作

IDT が送信するすべての情報は、*<device-tester-extract-location>*/results/*<execution-id>*/ フォルダの metrics.log ファイルにも口グが記録されます。口グファイルを表示すると、テスト実行中に収集された情報を確認できます。このファイルは、使用状況メトリックを収集することを選択した場合にのみ生成されます。

メトリクスの収集を無効にするために、追加のアクションを実行する必要はありません。AWS 認証情報を保管しなければ無効になります。AWS 認証情報を保管した場合は、この情報にアクセスする config.json ファイルを設定しないようにしてください。

AWS 認証情報を設定する

AWS アカウント をまだ持っていない場合は、作成する必要があります。AWS アカウント を既にお持ちの場合は、自分に代わって IDT が AWS に使用状況メトリクスを送信できるようにするために、必要なアクセス許可をアカウントに設定する必要があります。

ステップ 1: AWS アカウント を作成する

このステップでは、AWS アカウント を作成して設定します。AWS アカウント を既にお持ちの場合は、the section called “ステップ 2: IDT 用のアクセス許可を設定する” に進んでください。

AWS アカウント がない場合は、以下のステップを実行して作成します。

AWS アカウント にサインアップするには

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、通話呼び出しを受け取り、電話のキーパッドを用いて検証コードを入力するように求められます。

   AWS アカウント にサインアップすると、AWS アカウントのルートユーザー が作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティのベストプラクティスとして、管理ユーザーに管理アクセスを割り当て、ルートユーザーのみを使用してルートユーザーアクセスが必要なタスクを実行してください。

管理者ユーザーを作成するには、以下のいずれかのオプションを選択します。

管理者を管理する方法を 1 つ選択します	To	By	以下の操作も可能
IAM Identity Center 内 (推奨)	短期認証情報を使用して AWS にアクセスする。 これはセキュリティのベストプラクティスと一致しています。ベストプラクティスの詳細については、IAM ユーザーガイドの「IAM でのセキュリティのベストプラクティス」を参照してください。	AWS IAM Identity Center ユーザーガイドの「開始方法」の手順に従います。	AWS Command Line Interface ユーザーガイドの「AWS IAM Identity Center を使用するための AWS CLI の設定」に従って、プログラムによるアクセスを設定します。

管理者を管理する方法を 1 つ選択します	To	By	以下の操作も可能
IAM 内 (非推奨)	長期認証情報を使用して AWS にアクセスする。	IAM ユーザーガイドの「最初の IAM 管理者のユーザーおよびグループの作成」の手順に従います。	IAM ユーザーガイドの「IAM ユーザーのアクセスキーの管理」に従って、プログラムによるアクセスを設定します。

ステップ 2: IDT 用のアクセス許可を設定する

このステップでは、IDT がテストを実行して IDT 使用状況データを収集するために使用するアクセス許可を設定します。AWS Management Console または AWS Command Line Interface (AWS CLI) を使用して、IDT 用の IAM ポリシーとユーザーを作成し、そのユーザーにポリシーをアタッチできます。

- IDT 用のアクセス許可を設定するには (コンソール)
- IDT 用のアクセス許可を設定するには (AWS CLI)

IDT 用のアクセス許可を設定するには (コンソール)

コンソールを使用して IDT for AWS IoT Greengrass 用のアクセス許可を設定するには、次のステップに従ってください。

1. IAM コンソールにサインインします。
2. 特定のアクセス許可を持つロールを作成するためのアクセス許可を付与するカスタマー管理ポリシーを作成します。

   a. ナビゲーションペインで、[Policies] (ポリシー) を選択し、次に [Create policy] (ポリシーの作成) を選択します。

   b. JSON タブで、プレースホルダーコンテンツを以下のポリシーに置き換えます。

```
{
    "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot-device-tester:SendMetrics"
            ],
            "Resource": "*"
        }
    ]
}
```

c. [Review policy] (ポリシーの確認) を選択します。

d. [Name (名前)] に **`IDTUsageMetricsIAMPermissions`** と入力します。[概要] で、ポリシーによって付与されたアクセス許可を確認します。

e. [Create policy] (ポリシーを作成) を選択します。

3. IAM ユーザーを作成し、ユーザーにアクセス許可をアタッチします。

a. IAM ユーザーを作成します。IAM ユーザーガイドの IAM ユーザーの作成 (コンソール) のステップ 1 ~ 5 に従います。IAM ユーザーを作成済みの場合は、次のステップに進んでください。

b. アクセス許可を IAM ユーザーにアタッチします。

i. [Set permissions] ページで、[Attach existing policies to user directly] を選択します。

ii. 前のステップで作成した IDTGreengrassIAMPermissions ポリシーを検索します。チェックボックスをオンにします。

c. [Next: Tags] (次へ: タグ) を選択します。

d. [Next: Review] (次へ: レビュー) を選択して、選択内容の概要を表示します。

e. [Create user] (ユーザーの作成) を選択します。

f. ユーザーのアクセスキー (アクセスキー ID とシークレットアクセスキー) を表示するには、パスワードとアクセスキーの横にある [Show] (表示) を選択します。アクセスキーを保存するには、[Download .csv] を選択し、安全な場所にファイルを保存します。この情報を後で使用して、AWS 認証情報ファイルを設定します。

IDT 用のアクセス許可を設定するには (AWS CLI)

AWS CLI を使用して IDT for AWS IoT Greengrass 用のアクセス許可を設定するには、次のステップに従ってください。

1. コンピュータに AWS CLI がまだインストールされていない場合は、インストールして設定します。AWS Command Line Interface ユーザーガイドの AWS CLI のインストールのステップに従います。

   > **Note**
   >
   > AWS CLI は、コマンドラインシェルから AWS のサービスとやり取りするために使用できるオープンソースツールです。

2. IDT と AWS IoT Greengrass ロールを管理するためのアクセス許可を付与する、次のカスタマー管理ポリシーを作成します。

   Linux or Unix

   ```
   aws iam create-policy --policy-name IDTUsageMetricsIAMPermissions --policy-
   document '{
       "Version": "2012-10-17",
       "Statement": [
           {
               "Effect": "Allow",
               "Action": [
                   "iot-device-tester:SendMetrics"
               ],
               "Resource": "*"
           }
       ]
   }'
   ```

   Windows command prompt

   ```
   aws iam create-policy --policy-name IDTUsageMetricsIAMPermissions --policy-
   document
                                           '{\"Version\": \"2012-10-17\",
    \"Statement\": [{\"Effect\": \"Allow\", \"Action\": [\"iot-device-
   tester:SendMetrics\"], \"Resource\": \"*\"}]}'
   ```

> **Note**
>
> このステップには、Linux、macOS、または Unix のターミナルコマンドとは異なる JSON 構文を使用するため、Windows コマンドプロンプトの例が含まれています。

PowerShell

```
aws iam create-policy --policy-name IDTUsageMetricsIAMPermissions --policy-
document '{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot-device-tester:SendMetrics"
            ],
            "Resource": "*"
        }
    ]
}'
```

3. IAM ユーザーを作成し、IDT for AWS IoT Greengrass に必要なアクセス許可をアタッチします。

   a. IAM ユーザーを作成します。

   ```
   aws iam create-user --user-name user-name
   ```

   b. 作成した IDTUsageMetricsIAMPermissions ポリシーを IAM ユーザーにアタッチします。*user-name* を IAM ユーザー名に置き換え、コマンドの *<account-id>* を AWS アカウント の ID に置き換えます。

   ```
   aws iam attach-user-policy --user-name user-name --policy-arn
    arn:aws:iam::<account-id>:policy/IDTGreengrassIAMPermissions
   ```

4. ユーザーのシークレットアクセスキーを作成します。

```
aws iam create-access-key --user-name user-name
```

この出力は安全な場所に保存してください。この情報を後で使用して、AWS 認証情報ファイルを設定します。

IDT に AWS 認証情報を提供する

IDT に AWS 認証情報へのアクセスと、AWS へのメトリクスの送信を許可するには、次の手順を実行します。

1. 環境変数として、または認証情報ファイル内に、IAM ユーザーの AWS 認証情報を保管します。

   a. 環境変数を使用するには、次のコマンドを実行します。

      Linux or Unix

      ```
      export AWS_ACCESS_KEY_ID=access-key
      export AWS_SECRET_ACCESS_KEY=secret-access-key
      ```

      Windows Command Prompt (CMD)

      ```
      set AWS_ACCESS_KEY_ID=access-key
      set AWS_SECRET_ACCESS_KEY=secret-access-key
      ```

      PowerShell

      ```
      $env:AWS_ACCESS_KEY_ID="access-key"
      $env:AWS_SECRET_ACCESS_KEY="secret-access-key"
      ```

   b. 認証情報ファイルを使用するには、`~/.aws/credentials` ファイルに次の情報を追加します。

      ```
      [profile-name]
      aws_access_key_id=access-key
      aws_secret_access_key=secret-access-key
      ```

2. `config.json` ファイルの `auth` セクションを設定します。詳細については、「(オプション) config.json の設定」を参照してください。

IDT for AWS IoT Greengrass V2 のトラブルシューティング

IDT for AWS IoT Greengrass V2 は、エラーの種類に基づいて、エラーをさまざまな場所に書き込みます。IDT はコンソール、ログファイル、テストレポートにエラーを書き込みます。

エラーをどこで探せばよいか

テスト実行中はエラーの概要がコンソールに表示され、テストがすべて完了すると、失敗したテストの概要が表示されます。`awsiotdevicetester_report.xml` には、テストが失敗する原因となったすべてのエラーの概要が含まれます。IDT はテスト実行ごとのログファイルを、テスト実行用の UUID を持つディレクトリに保存します。これはテスト実行中はコンソールに表示されます。

IDT テストログのディレクトリは *<device-tester-extract-location>*`/results/`*<execution-id>*`/logs/` です。このディレクトリには、テーブルに表示されている次のファイルが含まれています。これは、デバッグ時に便利です。

File	説明
`test_manager.log`	テストの実行中にコンソールに書き込まれたログ。このファイルの最後にある結果の概要には、失敗したテストのリストが含まれます。 失敗に関する情報は、このファイルの警告ログとエラーログで確認できます。
test-group-id `/`*test-case-id* `/`*test-name* `.log`	テストグループ内の特定のテストの詳細なログ。Greengrass コンポーネントをデプロイするテストの場合、テストケースログファイルは `greengrass-test-run.log` と呼ばれます。
test-group-id `/`*test-case-id* `/greengrass.log`	AWS IoT Greengrass Core ソフトウェアの詳細なログ。IDT は、デバイスに AWS IoT Greengrass Core ソフトウェアをインストールするテストを実行する際は、テスト対象のデバイスからこのファイルをコピーします。このログファイルのメッセージの詳細につい

File	説明
	ては、「トラブルシューティング AWS IoT Greengrass V2」を参照してください。
test-group-id /*test-case-id*/*component-name* .log	テスト実行中にデプロイされる Greengrass コンポーネントの詳細なログ。IDT は、特定のコンポーネントをデプロイするテストを実行するときに、テスト対象のデバイスからコンポーネントログファイルをコピーします。各コンポーネントログファイルの名前は、デプロイされたコンポーネントの名前に対応します。このログファイルのメッセージの詳細については、「トラブルシューティング AWS IoT Greengrass V2」を参照してください。

IDT for AWS IoT Greengrass V2 エラーの解決

IDT for AWS IoT Greengrass を実行する前に、正しい設定ファイルを所定の場所に配置してください。解析エラーや設定エラーが表示される場合は、まず環境に適した設定テンプレートを見つけて使用します。

それでも問題が解決されない場合は、次のデバッグプロセスを参照してください。

トピック

- エイリアスの解決エラー
- 競合エラー
- テストを開始できなかったエラー
- Docker 認定イメージが存在するエラー
- 認証情報を読み込めませんでした
- 事前インストールされた Greengrass で発生する Guice エラー
- 無効な署名の例外
- 機械学習認定エラー
- オープンテストフレームワーク (OTF) のデプロイ失敗
- 解析エラー

- アクセス拒否エラー
- 認定レポート生成エラー
- 必須パラメータが見つからないエラー
- macOS でのセキュリティ例外
- SSH 接続エラー
- ストリームマネージャー認定エラー
- タイムアウトエラー
- バージョンチェックエラー

エイリアスの解決エラー

カスタムテストスイートを実行すると、コンソールおよび `test_manager.log` で以下のエラーが表示される場合があります。

```
Couldn't resolve placeholders: couldn't do a json lookup: index out of range
```

このエラーは、IDT テストオーケストレーターで設定されたエイリアスが正しく解決されない場合、または解決された値が設定ファイル内に存在しない場合に発生します。このエラーを解決するには、`device.json` および `userdata.json` にテストスイートに必要な正しい情報を記載するようにしてください。AWS IoT Greengrass の認定に必要な設定の詳細については、「AWS IoT Greengrass 認定スイートを実行するための IDT 設定を設定する」を参照してください。

競合エラー

AWS IoT Greengrass 認定スイートを複数のデバイスで同時に実行すると、以下のエラーが表示される場合があります。

```
ConflictException: Component [com.example.IDTHelloWorld : 1.0.0] for account [account-
id] already exists with state: [DEPLOYABLE] { RespMetadata: { StatusCode: 409,
 RequestID: “id” }, Message_: “Component [com.example.IDTHelloWorld : 1.0.0] for
 account [account-id] already exists with state: [DEPLOYABLE]” }
```

AWS IoT Greengrass 認定スイートでは、テストの同時実行は現時点ではサポートされていません。認定スイートは、デバイスごとに順番に実行してください。

テストを開始できなかったエラー

テストを開始しようとしたときに発生した障害を示すエラーが表示される場合があります。考えられる原因にはさまざまなものがあるため、以下を実行します。

- 実行コマンド内のプール名が実際に存在することを確認します。IDT は、プール名を `device.json` ファイルから直接参照します。
- プール内のデバイスの設定パラメータが正しいことを確認します。

Docker 認定イメージが存在するエラー

Docker アプリケーションマネージャーの認定テストでは、テスト対象のデバイスの認定に、Amazon ECR の `amazon/amazon-ec2-metadata-mock` コンテナイメージが使用されます。

テスト対象のデバイスの Docker コンテナにイメージがすでに存在する場合は、以下のエラーが表示される場合があります。

```
The Docker image amazon/amazon-ec2-metadata-mock:version already exists on the device.
```

以前このイメージをダウンロードして、デバイスで `amazon/amazon-ec2-metadata-mock` コンテナを実行していた場合は、認定テストを実行する前に、テスト対象のデバイスからこのイメージを削除してください。

認証情報を読み込めませんでした

Windows デバイスをテストするときに、テスト対象のデバイスへの接続に使用するユーザーがそのデバイスの認証情報マネージャーで設定されていないと、`greengrass.log` ファイルに `Failed to read credential` エラーが表示される場合があります。

このエラーを解決するには、テスト対象のデバイスの認証情報マネージャーで IDT ユーザーのユーザーとパスワードを設定します。

詳細については、「Windows デバイスのユーザー認証情報を設定する」を参照してください。

事前インストールされた Greengrass で発生する Guice エラー

事前インストールされた Greengrass を使用して IDT を実行している際、`Guice` または `ErrorInCustomProvider` というエラーが発生した場合は、ファイル `userdata.json` 内の `InstalledDirRootOnDevice` が、Greengrass のインストールフォルダに設定さ

れているかどうかを確認します。IDT は <InstallationDirRootOnDevice>/config/effectiveConfig.yaml の下の effectiveConfig.yaml ファイルをチェックします。

詳細については、「Windows デバイスのユーザー認証情報を設定する」を参照してください。

無効な署名の例外

Lambda 認定テストを実行するとき、IDT ホストマシンにネットワークアクセスの問題が発生すると、invalidsignatureexception エラーが発生します。ルーターをリセットして、テストを再度実行してください。

機械学習認定エラー

機械学習 (ML) 認定テストを実行するとき、AWS が提供する ML コンポーネントをデプロイするための要件をデバイスが満たしていないと、認定エラーが発生する場合があります。ML 認定エラーのトラブルシューティングを行うには、以下の手順を実行します。

- テスト実行中にデプロイされたコンポーネントのコンポーネントログで、エラーの詳細を確認します。コンポーネントログは *<device-tester-extract-location>*/results/*<execution-id>*/logs/*<test-group-id>* ディレクトリにあります。
- -Dgg.persist=installed.software の引数を失敗したテストケースの test.json ファイルに追加します。test.json ファイルは *<device-tester-extract-location>*/tests/GGV2Q_*version* directory. にあります。

オープンテストフレームワーク (OTF) のデプロイ失敗

OTF テストでデプロイが失敗しなかった場合は、TempResourcesDirOnDevice および InstallationDirRootOnDevice の親フォルダーに対するアクセス許可に原因があると考えられます。このフォルダーへのアクセス許可を適切に作成するには、次のコマンドを実行します。*folder-name* は、実際の親フォルダーの名前に置き換えます。

```
sudo chmod 755 folder-name
```

解析エラー

JSON 設定のタイプミスは、解析エラーにつながることがあります。ほとんどの場合、JSON ファイルで括弧やカンマ、引用符を忘れたことが原因です。IDT は、JSON 検証を行い、デバッグ情報を出力します。エラーが発生した行、構文エラーの行番号と列番号が出力されます。この情報だけで

エラーの修正が可能なはずですが、それでもエラーを特定できない場合は、IDE、テキストエディタ (Atom、Sublime など)、またはオンラインツール (JSONLint など) を使って手動で検証できます。

アクセス拒否エラー

IDT は、テスト対象デバイスのさまざまなディレクトリやファイルに対してオペレーションを実行します。一部のオペレーションにはルートアクセスが必要です。これらのオペレーションを自動化するには、パスワードを入力することなく、IDT で sudo を使用してコマンドを実行する必要があります。

パスワードを入力することなく、sudo にアクセスを許可するには、以下の手順を実行します。

> **Note**
>
> `user` および `username` は、テスト対象デバイスにアクセスするために IDT で使用する SSH ユーザーを指します。

1. SSH ユーザーを sudo グループに追加するには sudo usermod -aG sudo ***<ssh-username>*** を使用します。
2. サインアウトし、再度サインインして、変更を反映します。
3. `/etc/sudoers` ファイルを開き、ファイルの末尾に次の行を追加します: *<ssh-username>* `ALL=(ALL) NOPASSWD: ALL`

   > **Note**
   >
   > ベストプラクティスとして、`/etc/sudoers` を編集するときは sudo visudo を使用することをお勧めします。

認定レポート生成エラー

IDT は AWS IoT Greengrass V2 認定スイート (GGV2Q) の *major*.*minor* の最新 4 バージョンをサポートしており、これにより、AWS Partner Device Catalog にデバイスを登録するために AWS Partner Network に送信する認定レポートを生成できます。以前のバージョンの認定スイートでは、認定レポートは生成されません。

サポートポリシーについてご質問がある場合は、 AWS Support までお問い合わせください。

必須パラメータが見つからないエラー

IDT が新しい機能を追加すると、設定ファイルに変更が加えられる可能性があります。古い設定ファイルを使用すると、設定が破損する可能性があります。このような場合は、`/results/<execution-id>/logs` にある `<test_case_id>.log` ファイルに、すべての不足しているパラメータが明確に示されています。また、IDT では、JSON 設定ファイルのスキーマを検証し、サポートされている最新のバージョンが使用されているか検証します。

macOS でのセキュリティ例外

macOS ホストコンピュータで IDT を実行すると、IDT の実行がブロックされます。IDT を実行するには、セキュリティ例外を IDT ランタイム機能の一部である実行可能ファイルに付与します。ホストコンピュータに警告メッセージが表示されたら、該当する実行ファイルごとに次の操作を行います。

セキュリティ例外を IDT 実行可能ファイルに付与するには

1. macOS コンピュータの Apple メニューで、[System Preferences] (システム環境設定) を開きます。
2. [Security & Privacy] (セキュリティとプライバシー) を選択し、[General] (一般) タブでロックアイコンを選択して、セキュリティ設定を変更します。
3. ブロックされた `devicetester_mac_x86-64` の場合は、メッセージ `"devicetester_mac_x86-64" was blocked from use because it is not from an identified developer.` を探して [Allow Anyway] (すべてのアプリケーションを許可) を選択します。
4. 関連するすべての実行可能ファイルを完了するまで、IDT テストを再開します。

SSH 接続エラー

IDT からテスト対象デバイスに接続できない場合は、接続エラーのログが `/results/<execution-id>/logs/<test-case-id>.log` に記録されます。SSH メッセージは、このログファイルの上部に表示されます。テスト対象デバイスへの接続は IDT が実行する最初のオペレーションの 1 つであるためです。

ほとんどの Windows 設定では、PuTTy ターミナルアプリケーションを使用して Linux ホストに接続します。このアプリケーションは、標準 PEM プライベートキーファイルを PPK と呼ばれる独自の Windows 形式に変換することを要求します。`device.json` ファイルで SSH を設定する

場合は、PEM ファイルを使用してください。PPK ファイルを使用する場合、IDT では AWS IoT Greengrass デバイスとの SSH 接続を作成できず、テストを実行することができません。

IDT v4.4.0 以降、テスト対象のデバイスで SFTP を有効にしていない場合、ログファイルに次のエラーが表示される場合があります。

```
SSH connection failed with EOF
```

このエラーを解決するには、デバイスで SFTP を有効にします。

ストリームマネージャー認定エラー

ストリームマネージャー認定テストを実行すると、com.aws.StreamManagerExport.log ファイルに以下のエラーが表示されることがあります。

```
Failed to upload data to S3
```

このエラーは、ストリームマネージャーがデバイスで ~/root/.aws/credentials ファイルの AWS 認証情報を使用し、IDT がテスト対象のデバイスにエクスポートする環境認証情報が使用されていないとき発生します。この問題を回避するには、デバイスの credentials ファイルを削除し、認定テストを再実行してください。

タイムアウトエラー

各テストのタイムアウトを長くするには、各テストのタイムアウトのデフォルト値に適用されるタイムアウト乗数を指定します。このフラグに設定された値はすべて、1.0 以上である必要があります。

タイムアウトの乗数を使用するには、テストの実行時に --timeout-multiplier フラグを使用します。例:

```
./devicetester_linux run-suite --suite-id GGV2Q_1.0.0 --pool-id DevicePool1 --timeout-
multiplier 2.5
```

詳細については、run-suite --help を実行してください。

一部のタイムアウトエラーは、設定の問題により IDT テストケースを完了できない場合に発生します。このようなエラーは、タイムアウト乗数を増やしても解決することはできません。テスト実行のログを使用して、基本的な設定の問題に対するトラブルシューティングを行ってください。

- MQTT または Lambda コンポーネントのログにAccess denied エラーが含まれる場合、Greengrass インストールフォルダに適切なファイルのアクセス許可が与えられていない可能

性があります。userdata.json ファイルで定義したインストールパス内のフォルダごとに、以下のコマンドを実行します。

```
sudo chmod 755 folder-name
```

- Greengrass ログに Greengrass CLI のデプロイが完了していないことが示されている場合は、以下の手順を実行します。
  - テスト対象デバイスに bash がインストールされていることを確認します。
  - userdata.json ファイルに GreengrassCliVersion 設定パラメータが含まれている場合、それを削除します。IDT v4.1.0 以降のバージョンでは、このパラメータは廃止されています。詳細については、「userdata.json を設定する」を参照してください。
- Lambda デプロイテストが「Validating Lambda publish: time out」(Lambda の発行を検証しています: タイムアウト) というエラーメッセージで失敗し、テストログファイル(idt-gg2-lambda-function-idt-*<resource-id>*.log) に Error: Could not find or load main class com.amazonaws.greengrass.runtime.LambdaRuntime. というエラーが表示された場合は、次を実行します。
  - userdata.json ファイルで InstallationDirRootOnDevice のために使用されたフォルダを検証します。
  - デバイスに正しいユーザー許可が設定されていることを確認してください。詳細については、「デバイスに対するユーザーのアクセス許可を設定する」を参照してください。

バージョンチェックエラー

IDT ユーザーの AWS ユーザー認証情報に、必要な IAM アクセス権限が与えられていない場合、IDT は以下のエラーを発行します。

```
Failed to check version compatibility
```

必要な IAM アクセス権限を持たない AWS ユーザー。

AWS IoT Device Tester の のサポートポリシー AWS IoT Greengrass

AWS IoT Device Tester for AWS IoT Greengrass は、 AWS IoT Greengrass Device AWS Partner Catalog に含めるデバイスを検証および認定するために使用されるテスト自動化ツールです。デバイ

スのテストまたは認定 AWS IoT Device Tester には、 AWS IoT Greengrass および の最新バージョンを使用することをお勧めします。

サポートされている のバージョンごとに、少なくとも 1 つのバージョンの AWS IoT Device Tester を使用できます AWS IoT Greengrass。のサポートされているバージョンについては AWS IoT Greengrass、「Greengrass nucleus バージョン」を参照してください。のサポートされているバージョンについては AWS IoT Device Tester、「」を参照してくださいAWS IoT Device Tester for AWS IoT Greengrass V2 のサポートされているバージョン。

また、 AWS IoT Greengrass および のサポートされている任意のバージョンを使用して AWS IoT Device Tester 、デバイスをテストまたは認定することもできます。サポートされていないバージョンの は引き続き使用できますが AWS IoT Device Tester、これらのバージョンはバグ修正や更新を受け取りません。サポートポリシーについてご質問がある場合は、 AWS Support までお問い合わせください。

Greengrass ベースの IoT ソリューション

Vatech の Everyware GreenEdge は のプレビューリリースに含まれAWS IoT Greengrassており、変更される可能性があります。このソリューションは、AWS ではサポートされていません。このデバイスに関する問題については、Euroratech に連絡する必要があります。

AWS IoT Greengrass は、Greengrass のインストール体験を最適化するための パートナーからのソリューションを提供します。以下は、Euroratech と提携して提供AWSしているソリューションです。このソリューションには、 AWS IoT Greengrass Core エッジランタイムと追加機能がプリインストールされています。

テルテック語

AWS は、Euroratech と提携して、 AWS IoT Greengrass Core ソフトウェアがプリインストールされているデバイスを探しているお客様に IoT ソリューションを提供します。DCRtech の Everyware GreenEdge は、 によって事前設定および事前認定された IoT エッジソフトウェアですAWS。このソリューションでは、Greengrass と Vatech Everyware Software Framework (ESF) の機能を統合して、Modbus、OPC-UA クライアント/サーバー、S7、 TwinCAT、J1939, DNP3 Master/Outstation などのプロトコルアダプタを介して、顧客に広範な上限接続を提供します。このソリューションでは、 にデータを送信AWS クラウドし、すべての北境界AWSサービス (AWS IoT Core、、AWS IoT SiteWiseAWS IoT AnalyticsAmazon S3、Amazon Kinesis Video Streams など) に接続することもできます。このソリューションは、Euroratech のデバイス管理ソリューションである Everyware Cloud と組み合わせると、新しいゼロタッチプロビジョニングサービスを導入し、デバイスのオンボーディングと一括デプロイを簡素化します。

Vatech の詳細については、「Euroratech」を参照してください。

トラブルシューティング AWS IoT Greengrass V2

このセクションのトラブルシューティング情報と解決策を使用して、 の問題を解決してください AWS IoT Greengrass Version 2。

トピック

AWS IoT Greengrass Core ソフトウェアとコンポーネントのログを表示する

AWS IoT Greengrass Core ソフトウェアは、コアデバイスに関するリアルタイム情報を表示するために使用できるローカルファイルシステムにログを書き込みます。ログにログを書き込むようにコアデバイスを設定して CloudWatch 、コアデバイスをリモートでトラブルシューティングすることもできます。これらのログは、コンポーネント、デプロイ、およびコアデバイスの問題を特定するのに役立ちます。詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

AWS IoT Greengrass Core ソフトウェアの問題

AWS IoT Greengrass Core ソフトウェアの問題のトラブルシューティングを行います。

トピック

- コアデバイスをセットアップできない
- AWS IoT Greengrass Core ソフトウェアをシステムサービスとして起動できない
- nucleus をシステムサービスとしてセットアップできない
- AWS IoT Core に接続できない
- メモリ不足エラー
- Greengrass CLI をインストールできない
- User root is not allowed to execute
- com.aws.greengrass.lifecyclemanager.GenericExternalService: Could not determine user/group to run with
- Failed to map segment from shared object: operation not permitted
- Windows サービスのセットアップに失敗しました
- com.aws.greengrass.util.exceptions.TLSAuthException: Failed to get trust manager
- com.aws.greengrass.deployment.IotJobsHelper: No connection available during subscribing to Iot Jobs descriptions topic. Will retry in sometime
- software.amazon.awssdk.services.iam.model.IamException: The security token included in the request is invalid
- software.amazon.awssdk.services.iot.model.IotException: User: <user> is not authorized to perform: iot:GetPolicy
- Error: com.aws.greengrass.shadowmanager.sync.model.FullShadowSyncRequest: Could not execute cloud shadow get request
- Operation aws.greengrass#<operation> is not supported by Greengrass
- java.io.FileNotFoundException: <stream-manager-store-root-dir>/stream_manager_metadata_store (Permission denied)
- com.aws.greengrass.security.provider.pkcs11.PKCS11CryptoKeyService: Private key or certificate with label <label> does not exist
- software.amazon.awssdk.services.secretsmanager.model.SecretsManagerException: User: <user> is not authorized to perform: secretsmanager:GetSecretValue on resource: <arn>
- software.amazon.awssdk.services.secretsmanager.model.SecretsManagerException: Access to KMS is not allowed
- java.lang.NoClassDefFoundError: com/aws/greengrass/security/CryptoKeySpi
- com.aws.greengrass.security.provider.pkcs11.PKCS11CryptoKeyService: CKR_OPERATION_NOT_INITIALIZED

コアデバイスをセットアップできない

AWS IoT Greengrass Core ソフトウェアインストーラが失敗し、コアデバイスをセットアップできない場合は、ソフトウェアをアンインストールして再試行する必要がある場合があります。詳細については、「AWS IoT Greengrass Core ソフトウェアをアンインストールする」を参照してください。

AWS IoT Greengrass Core ソフトウェアをシステムサービスとして起動できない

AWS IoT Greengrass Core ソフトウェアが起動しない場合は、システムサービスログを確認して問題を特定します。よくある問題の 1 つは、Java が PATH 環境変数 (Linux) または PATH システム変数 (Windows) で使用できない場合です。

nucleus をシステムサービスとしてセットアップできない

このエラーは、 AWS IoT Greengrass Core ソフトウェアインストーラがシステムサービス AWS IoT Greengrass として設定できない場合に表示されることがあります。Linux デバイスでは、このエラーは通常、コアデバイスにsystemdinit システムがない場合に発生します。インストーラは、システムサービスのセットアップに失敗した場合でも、 AWS IoT Greengrass Core ソフトウェアを正常にセットアップできます。

次のいずれかを行います。

- AWS IoT Greengrass Core ソフトウェアをシステムサービスとして設定して実行します。 AWS IoT Greengrass のすべての機能を使用するには、ソフトウェアをシステムサービスとして設定する必要があります。systemd をインストール、または別の init システムを使用できます。詳細については、「Greengrass nucleus をシステムサービスとして設定する」を参照してください。
- システムサービスを使用せずに AWS IoT Greengrass Core ソフトウェアを実行します。インストーラが Greengrass ルートフォルダでセットアップしたローダースクリプトを使用して、ソフトウェアを実行できます。詳細については、「システムサービスを使用せずに AWS IoT Greengrass Core ソフトウェアを実行します。」を参照してください。

AWS IoT Core に接続できない

このエラーは、例えば、 AWS IoT Greengrass Core ソフトウェアが に接続 AWS IoT Core してデプロイジョブを取得できない場合に表示されることがあります。以下の操作を実行します。

- コアデバイスがインターネットと に接続できることを確認します AWS IoT Core。デバイスが接続する AWS IoT Core エンドポイントの詳細については、「」を参照してくださいAWS IoT Greengrass Core ソフトウェアを設定する。
- コアデバイスの AWS IoT モノが、`iot:Connect`、、`iot:Publish`、`iot:Receive`および の`iot:Subscribe`アクセス許可を付与する証明書を使用していることを確認します。
- コアデバイスが [network proxy] (ネットワークプロキシ)を使用している場合、コアデバイスに [device role] (デバイスロール)があり、そのロールで `iot:Connect`、`iot:Publish`、`iot:Receive`、`iot:Subscribe` のアクセス許可が付与されていることを確認してください。

メモリ不足エラー

このエラーは通常、デバイスに Java ヒープにオブジェクトを割り当てるだけの十分なメモリがない場合に発生します。メモリが制限されたデバイスでは、メモリ割り当てを制御するために最大ヒープサイズを指定する必要がある場合があります。詳細については、「JVM オプションでメモリ割り当てを制御する」を参照してください。

Greengrass CLI をインストールできない

AWS IoT Greengrass Core のインストールコマンドで `--deploy-dev-tools`引数を使用すると、次のコンソールメッセージが表示される場合があります。

```
Thing group exists, it could have existing deployment and devices, hence NOT creating
 deployment for Greengrass first party dev tools, please manually create a deployment
 if you wish to
```

これは、コアデバイスが既存のデプロイを持つモノのグループのメンバーであるため、Greengrass CLI コンポーネントがインストールされていない場合に発生します。このメッセージが表示された場合は、Greengrass CLI コンポーネント (`aws.greengrass.Cli`) をデバイスに手動でデプロイして、Greengrass CLI をインストールできます。詳細については、「Greengrass CLI のインストール」を参照してください。

User root is not allowed to execute

このエラーは、 AWS IoT Greengrass Core ソフトウェアを実行するユーザー (通常は `root`) に、ユーザーおよびグループsudoで実行するアクセス許可がない場合に表示されることがあります。デフォルトの `ggc_user` システムユーザーの場合、このエラーは次のようになります。

```
Sorry, user root is not allowed to execute <command> as ggc_user:ggc_group.
```

/etc/sudoers ファイルで、他のグループとして sudo を実行する権限がユーザーに与えられていることを確認してください。/etc/sudoers のユーザーの権限は、次の例のようになります。

```
root    ALL=(ALL:ALL) ALL
```

com.aws.greengrass.lifecyclemanager.GenericExternalService: Could not determine user/group to run with

このエラーは、コアデバイスがコンポーネントを実行しようとしたときに、Greengrass nucleus でコンポーネントの実行に使用するデフォルトのシステムユーザーが指定されていない場合に表示されることがあります。

この問題を解決するには、コンポーネントを実行するデフォルトのシステムユーザーを指定するように Greengrass nucleus を設定します。詳細については、コンポーネントを実行するユーザーを設定するおよびデフォルトのコンポーネントユーザーを設定するを参照してください。

Failed to map segment from shared object: operation not permitted

このエラーは、/tmpフォルダが アクセスnoexec許可でマウントされているために AWS IoT Greengrass Core ソフトウェアが起動しなかった場合に表示されることがあります。AWS Common Runtime (CRT) ライブラリでは、デフォルトで /tmp フォルダを使用します。

次のいずれかを行います。

- 次のコマンドを実行して、exec アクセス許可で /tmp フォルダを再マウントし、再試行します。

  ```
  sudo mount -o remount,exec /tmp
  ```

- Greengrass nucleus v2.5.0 以降を実行している場合は、JVM オプションを設定して、 AWS CRT ライブラリが使用するフォルダを変更できます。パラメータは、Greengrass nucleus コンポーネント設定のデプロイまたは AWS IoT Greengrass Core ソフトウェアのインストール時に指定できます。*/path/to/use* を AWS CRT ライブラリが使用できるフォルダへのパスに置き換えます。

  ```
  {
  ```

```
  "jvmOptions": "-Daws.crt.lib.dir=\"/path/to/use\""
}
```

Windows サービスのセットアップに失敗しました

Microsoft Windows 2016 デバイスに AWS IoT Greengrass Core ソフトウェアをインストールすると、このエラーが表示されることがあります。 AWS IoT Greengrass Core ソフトウェアは Windows 2016 ではサポートされていません。サポートされているオペレーティングシステムのリストについては、「」を参照してくださいサポートされているプラットフォーム。

Windows 2016 を使用する必要がある場合は、次の操作を実行します。

1. ダウンロードした AWS IoT Greengrass Core インストールアーカイブを解凍する
2. `Greengrass` ディレクトリで、`bin/greengrass.xml.template` ファイルを開きます。
3. `</service>` タグの直前に、ファイルの末尾に `<autoRefresh>` タグを追加します。

```
  </log>
  <autoRefresh>false</autoRefresh>
</service>
```

com.aws.greengrass.util.exceptions.TLSAuthException: Failed to get trust manager

このエラーは、ルート認証機関 (CA) ファイルなしで AWS IoT Greengrass Core ソフトウェアをインストールすると表示されることがあります。

```
2022-06-05T10:00:39.556Z [INFO] (main) com.aws.greengrass.lifecyclemanager.Kernel:
 service-loaded. {serviceName=DeploymentService}
2022-06-05T10:00:39.943Z [WARN] (main)
 com.aws.greengrass.componentmanager.ClientConfigurationUtils: configure-greengrass-
mutual-auth. Error during configure greengrass client mutual auth. {}
com.aws.greengrass.util.exceptions.TLSAuthException: Failed to get trust manager
```

`rootCaPath` パラメータがある有効なルート CA ファイルを、インストーラに与える設定ファイルに指定したことを確認します。詳細については、「AWS IoT Greengrass Core ソフトウェアをインストールします。」を参照してください。

com.aws.greengrass.deployment.IotJobsHelper: No connection available during subscribing to Iot Jobs descriptions topic. Will retry in sometime

この警告メッセージは、コアデバイスが に接続 AWS IoT Core してデプロイジョブ通知をサブスクライブできない場合に表示されることがあります。以下の操作を実行します。

- コアデバイスがインターネットに接続され、設定した AWS IoT データエンドポイントに到達できることを確認します。コアデバイスが使用するエンドポイントの詳細については、「プロキシまたはファイアウォールを介したデバイストラフィックを許可する」を参照してください。
- Greengrass ログをチェックして、他の根本原因を明らかにする他のエラーがないか確認します。

software.amazon.awssdk.services.iam.model.IamException: The security token included in the request is invalid

このエラーは、自動プロビジョニングを使用して AWS IoT Greengrass Core ソフトウェアをインストールし、インストーラが無効な AWS セッショントークンを使用しているときに表示されることがあります。以下の操作を実行します。

- 一時的なセキュリティ認証情報を使用する場合は、セッショントークンが正しいこと、および完全なセッショントークンをコピーして貼り付けていることを確認します。
- 長期的なセキュリティ認証情報を使用する場合は、以前に一時的な認証情報を使用した時点のセッショントークンがデバイスにないことを確認します。以下の操作を実行します。
  1. 次のコマンドを実行して、セッショントークンの環境変数の設定を解除します。

     Linux or Unix

     ```
     unset AWS_SESSION_TOKEN
     ```

     Windows Command Prompt (CMD)

     ```
     set AWS_SESSION_TOKEN=
     ```

     PowerShell

     ```
     Remove-Item Env:\AWS_SESSION_TOKEN
     ```

2. AWS 認証情報ファイル に~/.aws/credentialsセッショントークン が含まれているかどうかを確認しますaws_session_token。もしそうなら、ファイルからその行を削除します。

```
aws_session_token = AQoEXAMPLEH4aoAH0gNCAPyJxz4BlCFFxWNE1OPTgk5TthT
+FvwqnKwRc0IfrRh3c/LTo6UDdyJwOOvEVPvLXCrrrUtdnniCEXAMPLE/
IvU1dYUg2RVAJBanLiHb4IgRmpRV3zrkuWJOgQs8IZZaIv2BXIa2R4Olgk
```

AWS 認証情報を提供せずに AWS IoT Greengrass Core ソフトウェアをインストールすることもできます。詳細については、手動リソースプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールするまたはAWS IoT フリートプロビジョニングで AWS IoT Greengrass Core ソフトウェアをインストールするを参照してください。

software.amazon.awssdk.services.iot.model.IotException: User: <user> is not authorized to perform: iot:GetPolicy

このエラーは、自動プロビジョニングを使用して AWS IoT Greengrass Core ソフトウェアをインストールし、インストーラが必要なアクセス許可を持たない AWS 認証情報を使用する場合に表示されることがあります。必要なアクセス許可の詳細については、インストーラがリソースをプロビジョニングするための最小限の IAM ポリシー を参照してください。

認証情報の IAM アイデンティティに対するアクセス許可を確認し、不足していれば必要なアクセス許可を IAM アイデンティティに付与します。

Error: com.aws.greengrass.shadowmanager.sync.model.FullShadowSyncRequest: Could not execute cloud shadow get request

このエラーは、シャドウマネージャーコンポーネントを使用してデバイスシャドウを と同期 AWS IoT Coreする場合に表示されることがあります。HTTP 403 ステータスコードは、コアデバイスの AWS IoT ポリシーが を呼び出すアクセス許可を付与していないためにこのエラーが発生したことを示しますGetThingShadow。

```
com.aws.greengrass.shadowmanager.sync.model.FullShadowSyncRequest: Could not execute
 cloud shadow get request. {thing name=MyGreengrassCore, shadow name=MyShadow}
2021-07-14T21:09:02.456Z [ERROR] (pool-2-thread-109)
 com.aws.greengrass.shadowmanager.sync.SyncHandler: sync. Skipping sync request. {thing
 name=MyGreengrassCore, shadow name=MyShadow}
```

```
com.aws.greengrass.shadowmanager.exception.SkipSyncRequestException:
 software.amazon.awssdk.services.iotdataplane.model.IotDataPlaneException:
 null (Service: IotDataPlane, Status Code: 403, Request ID:
 f6e713ba-1b01-414c-7b78-5beb3f3ad8f6, Extended Request ID: null)
```

ローカルシャドウを と同期するには AWS IoT Core、コアデバイスの AWS IoT ポリシーで次のアクセス許可を付与する必要があります。

- `iot:GetThingShadow`
- `iot:UpdateThingShadow`
- `iot:DeleteThingShadow`

コアデバイスの AWS IoT ポリシーを確認し、不足している必要なアクセス許可を追加します。詳細については、次を参照してください。

- AWS IoT デベロッパーガイドの AWS IoT Core ポリシーアクション
- コアデバイスの AWS IoT ポリシーを更新する

Operation aws.greengrass#<operation> is not supported by Greengrass

このエラーは、カスタム Greengrass コンポーネントでプロセス間通信 (IPC) オペレーションを使用し、必要な AWSが提供するコンポーネントがコアデバイスにインストールされていない場合に表示されることがあります。

この問題を解決するには、コンポーネント recipe の依存関係として必要なコンポーネントを追加し、コンポーネントをデプロイするときに AWS IoT Greengrass Core ソフトウェアが必要なコンポーネントをインストールできるようにします。

- シークレット値を取得する – `aws.greengrass.SecretManager`
- ローカルシャドウとやり取りする – `aws.greengrass.ShadowManager`
- ローカルデプロイとコンポーネントを管理する – `aws.greengrass.Cli` v2.6.0 以降
- クライアントデバイスを認証して承認する – `aws.greengrass.clientdevices.Auth` v2.2.0 以降

java.io.FileNotFoundException: <stream-manager-store-root-dir>/stream_manager_metadata_store (Permission denied)

存在しない、または適切なアクセス許可を持たないルートフォルダを使用するようにストリームマネージャーを設定した場合に、ストリームマネージャーログファイル (`aws.greengrass.StreamManager.log`) にこのエラーが表示されることがあります。このフォルダの設定方法の詳細については、「ストリームマネージャー設定」を参照してください。

com.aws.greengrass.security.provider.pkcs11.PKCS11CryptoKeyService: Private key or certificate with label <label> does not exist

このエラーは、ハードウェアセキュリティモジュール (HSM) を使用するように AWS IoT Greengrass Core ソフトウェアを設定するときに指定するプライベートキーまたは証明書を PKCS#11 プロバイダコンポーネントが検出またはロードできない場合に発生します。以下の操作を実行します。

- Core ソフトウェアで使用するように設定したスロット、ユーザー PIN、オブジェクトラベルを使用して、プライベートキーと証明書が HSM AWS IoT Greengrass に保存されていることを確認します。
- プライベートキーと証明書が HSM で同じオブジェクトラベルを使用していることを確認します。
- HSM がオブジェクト ID をサポートしている場合は、プライベートキーと証明書が HSM で同じオブジェクト ID を使用していることを確認してください。

HSM のセキュリティトークンの詳細を照会する方法については、「HSM のマニュアル」を参照してください。セキュリティトークンのスロット、オブジェクトラベル、またはオブジェクト ID を変更する必要がある場合は、「HSM のマニュアル」を参照して、その方法を確認してください。

software.amazon.awssdk.services.secretsmanager.model.SecretsManagerExcep User: <user> is not authorized to perform: secretsmanager:GetSecretValue on resource: <arn>

このエラーは、シークレットマネージャーコンポーネントを使用して シー AWS Secrets Manager クレットをデプロイするときに発生する可能性があります。コアデバイスの [token exchange IAM role] (トークン交換 IAM ロール) がシークレットを取得するアクセス許可を付与しない場合、デプロイは失敗し、Greengrass のログにこのエラーが含まれます。

コアデバイスにシークレットのダウンロードを許可するには

1. コアデバイスのトークン交換ロールに `secretsmanager:GetSecretValue` アクセス許可を追加します。次に示すポリシーステートメントの例では、シークレットの値を取得するための権限を付与しています。

   ```
   {
       "Effect": "Allow",
       "Action": [
           "secretsmanager:GetSecretValue"
       ],
       "Resource": [
           "arn:aws:secretsmanager:us-west-2:123456789012:secret:MyGreengrassSecret-
   abcdef"
       ]
   }
   ```

   詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

2. コアデバイスにデプロイを再適用します。次のいずれかを行います。

   - 変更なしでデプロイを修正します。コアデバイスは、修正されたデプロイを受信すると、シークレットのダウンロードを再試行します。詳細については、「展開の改訂」を参照してください。
   - AWS IoT Greengrass Core ソフトウェアを再起動して、デプロイを再試行します。詳細については、「AWS IoT Greengrass Core ソフトウェアを実行する」を参照してください

   シークレットマネージャーがシークレットを正常にダウンロードすると、デプロイは成功します。

software.amazon.awssdk.services.secretsmanager.model.SecretsManagerExcep Access to KMS is not allowed

このエラーは、シークレットマネージャーコンポーネントを使用して、 キーで暗号化された AWS Secrets Manager シークレットを AWS Key Management Service デプロイするときに発生する可能性があります。コアデバイスのトークン交換 IAM ロールでシークレットを暗号化するアクセス許可が付与されない場合、デプロイは失敗し、Greengrass のログにこのエラーが含まれます。

この問題を解決するには、コアデバイスのトークン交換ロールに `kms:Decrypt` アクセス許可を追加します。詳細については、次を参照してください。

- 「AWS Secrets Manager ユーザーガイド」の「シークレット暗号化と復号」
- コアデバイスが AWS サービスを操作できるように認証する

java.lang.NoClassDefFoundError: com/aws/greengrass/security/CryptoKeySpi

このエラーは、ハードウェアセキュリティを備えた AWS IoT Greengrass Core ソフトウェアをインストールしようとして、ハードウェアセキュリティ統合をサポートしていない以前のバージョンの Greengrass nucleus を使用した場合に表示されることがあります。ハードウェアセキュリティ統合を使用するには、v2.5.3 以降の Greengrass nucleus を使用する必要があります。

com.aws.greengrass.security.provider.pkcs11.PKCS11CryptoKeyService: CKR_OPERATION_NOT_INITIALIZED

Core をシステムサービスとして実行するときに TPM2 AWS IoT Greengrass ライブラリを使用すると、このエラーが表示されることがあります。

このエラーは、 AWS IoT Greengrass Core systemd サービスファイルに PKCS#11 ストアの場所を提供する環境変数を追加する必要があることを示しています。

詳細については、PKCS#11 プロバイダ コンポーネントドキュメントの「要件」セクションを参照してください。

AWS IoT Greengrass クラウドの問題

次の情報を使用して、 AWS IoT Greengrass コンソールと API に関する問題のトラブルシューティングを行います。各エントリは、アクションを実行したときに表示されるエラーメッセージに対応します。

An error occurred (AccessDeniedException) when calling the CreateComponentVersion operation: User: arn:aws:iam::123456789012:user/<username> is not authorized to perform: null

このエラーは、 AWS IoT Greengrass コンソールから、または CreateComponentVersionオペレーションを使用してコンポーネントバージョンを作成するときに表示されることがあります。

このエラーは、recipe が有効な JSON または YAML でないことを示しています。recipe の構文を確認し、構文の問題を修正して、もう一度試してください。オンライン JSON または YAML 構文チェッカーを使用して、recipe の構文の問題を特定できます。

Invalid Input: Encountered following errors in Artifacts: {<s3ArtifactUri> = Specified artifact resource cannot be accessed}

このエラーは、 AWS IoT Greengrass コンソールから、または CreateComponentVersionオペレーションを使用してコンポーネントバージョンを作成するときに表示されることがあります。このエラーは、コンポーネント recipe の S3 アーティファクトが有効でないことを示しています。

以下の操作を実行します。

- S3 バケットがコンポーネントを作成する AWS リージョン のと同じ にあることを確認します。AWS IoT Greengrass は、コンポーネントアーティファクトのクロスリージョンリクエストをサポートしていません。
- アーティファクト URI が有効な S3 オブジェクト URL であることを確認し、その S3 オブジェクト URL にアーティファクトが存在することを確認します。
- に S3 オブジェクト URL のアーティファクトへのアクセス許可 AWS アカウント があることを確認します。

INACTIVE deployment status

必要な依存 AWS IoT ポリシーなしで ListDeployments API を呼び出すと、 `INACTIVE`デプロイステータスが表示されることがあります。正確なデプロイステータスを取得するには、必要なアクセス許可が必要です。依存アクションは、「 で定義されるアクション AWS IoT Greengrass V2」を参照し、 に必要なアクセス許可に従うと確認できます`ListDeployments`。必要な依存アクセス AWS IoT 許可がないと、デプロイのステータスは引き続き表示されますが、デプロイのステータスが不正確になる場合があります`INACTIVE`。

コアデバイスデプロイの問題

Greengrass コアデバイスでのデプロイの問題のトラブルシューティングを行います。各エントリは、コアデバイスに表示される可能性のあるログメッセージに対応します。

トピック

Error: com.aws.greengrass.componentmanager.exceptions.PackageDownloadExceptio Failed to download artifact

このエラーは、 AWS IoT Greengrass コアデバイスがデプロイを適用したときに Core ソフトウェアがコンポーネントアーティファクトのダウンロードに失敗した場合に表示されることがあります。このエラーの結果、デプロイは失敗します。

このエラーが発生した場合、ログにはスタックトレースも含まれ、特定の問題を識別するために使用できます。次の各エントリは、`Failed to download artifact` エラーメッセージのスタックトレースに表示される可能性のあるメッセージに対応しています。

トピック

- software.amazon.awssdk.services.s3.model.S3Exception: null (Service: S3, Status Code: 403, Request ID: null, ...)
- software.amazon.awssdk.services.s3.model.S3Exception: Access Denied (Service: S3, Status Code: 403, Request ID: <requestID>

software.amazon.awssdk.services.s3.model.S3Exception: null (Service: S3, Status Code: 403, Request ID: null, ...)

PackageDownloadException エラーには、次の場合にこのスタックトレースが含まれる場合があります。

- コンポーネントのアーティファクトは、コンポーネントの recipe で指定した S3 オブジェクト URL では利用できません。アーティファクトを S3 バケットにアップロードしたこと、アーティファクトの URI がバケット内のアーティファクトの S3 オブジェクト URL と一致していることを確認します。
- コアデバイスのトークン交換ロールは、 AWS IoT Greengrass Core ソフトウェアがコンポーネントの recipe で指定した S3 オブジェクト URL からコンポーネントアーティファクトをダウンロードすることを許可しません。トークン交換ロールで、アーティファクトが利用可能な S3 オブジェクト URL の `s3:GetObject` が許可されていることを確認します。

software.amazon.awssdk.services.s3.model.S3Exception: Access Denied (Service: S3, Status Code: 403, Request ID: <requestID>

このPackageDownloadException エラーには、コアデバイスに を呼び出すアクセス許可がない場合に、このスタックトレースが含まれることがありますs3:GetBucketLocation。このエラーメッセージには、次のメッセージも含まれています。

```
reason: Failed to determine S3 bucket location
```

コアデバイスのトークン交換ロールで、アーティファクトが利用可能な S3 バケットの s3:GetBucketLocation が許可されていることを確認します。

Error: com.aws.greengrass.componentmanager.exceptions.ArtifactChecksumMismatchI Integrity check for downloaded artifact failed. Probably due to file corruption.

このエラーは、 AWS IoT Greengrass コアデバイスがデプロイを適用したときに Core ソフトウェアがコンポーネントアーティファクトのダウンロードに失敗した場合に表示されることがあります。ダウンロードしたアーティファクトファイルのチェックサムが、コンポーネントの作成時に AWS IoT Greengrass 計算したチェックサムと一致しないため、デプロイは失敗します。

以下の操作を実行します。

- アーティファクトファイルが、それをホストする S3 バケット内で変更されたかどうかを確認します。コンポーネントの作成後にファイルが変更されている場合は、コアデバイスが想定している以前のバージョンに復元します。ファイルを以前のバージョンに復元できない場合、または新しいバージョンのファイルを使用する場合は、アーティファクトファイルを使用して新しいバージョンのコンポーネントを作成します。
- コアデバイスのインターネット接続を確認します。このエラーは、アーティファクトファイルがダウンロード中に破損した場合に発生する可能性があります。新しいデプロイを作成して、もう一度試してください。

Error: com.aws.greengrass.componentmanager.exceptions.NoAvailableComponentVers Failed to negotiate component <name> version with cloud and no local applicable version satisfying requirement <requirements>

このエラーは、コアデバイスで、そのコアデバイスのデプロイの要件を満たすコンポーネントバージョンが見つからない場合に表示されることがあります。コアデバイスは、 AWS IoT Greengrass サービス内およびローカルデバイス上のコンポーネントをチェックします。エラーメッセージには、各デプロイのターゲットと、そのコンポーネントのデプロイのバージョン要件が含まれます。デプロイターゲットはモノ、モノグループ、または LOCAL_DEPLOYMENT にすることができます。これは、コアデバイスのローカルデプロイを表します。

この問題は、次の状況に発生する可能性があります:

- コアデバイスは、競合するコンポーネントのバージョン要件を持つ複数のデプロイのターゲットです。例えば、コアデバイスは、com.example.HelloWorld コンポーネントを含む複数のデプロイのターゲットになる場合があり、一方のデプロイにはバージョン 1.0.0 が必要で、もう一方にはバージョン 1.0.1 が必要です。両方の要件を満たすコンポーネントを持つことは不可能であるため、デプロイは失敗します。
- コンポーネントバージョンが AWS IoT Greengrass サービスまたはローカルデバイスに存在しません。例えば、コンポーネントが削除された可能性があります。
- バージョン要件を満たすコンポーネントバージョンが存在しますが、コアデバイスのプラットフォームと互換性はありません。
- コアデバイスの AWS IoT ポリシーは、 アクセスgreengrass:ResolveComponentCandidates許可を付与しません。エラーログで Status Code: 403 を検索して、この問題を特定します。この問題を解決するには、コアデバイスの AWS IoT ポリシーに greengrass:ResolveComponentCandidates アクセス許可を追加します。詳細については、「AWS IoT Greengrass V2 コアデバイス向けの最低限の AWS IoT ポリシー」を参照してください。

この問題を解決するには、デプロイを修正して互換性のあるバージョンのコンポーネントを含めるか、互換性のないコンポーネントを削除します。クラウドデプロイを変更する方法の詳細については、「展開の改訂」を参照してください。ローカルデプロイを変更する方法の詳細については、AWS IoT Greengrass CLI デプロイ作成コマンドを参照してください。

software.amazon.awssdk.services.greengrassv2data.model.ResourceNotFoundE The latest version of Component <componentName> doesn't claim platform <coreDevicePlatform> compatibility

このエラーは、コンポーネントをコアデバイスにデプロイした場合に、コアデバイスのプラットフォームと互換性のあるプラットフォームがコンポーネントにリストされていないと表示されることがあります。次のいずれかを行います。

- コンポーネントがカスタム Greengrass コンポーネントの場合、コンポーネントを更新してコアデバイスと互換性を持たせることができます。コアデバイスのプラットフォームに一致する新しいマニフェストを追加するか、コアデバイスのプラットフォームに合わせて既存のマニフェストを更新します。詳細については、「AWS IoT Greengrass コンポーネントレシピのリファレンス」を参照してください。
- コンポーネントが によって提供されている場合は AWS、コンポーネントの別のバージョンがコアデバイスと互換性があるかどうかを確認します。互換性のあるバージョンがない場合は、AWS IoT Greengrass タグを使用して AWS re:Post でお問い合わせいただくか、AWS Support にお問い合わせください。

com.aws.greengrass.componentmanager.exceptions.PackagingException: The deployment attempts to update the nucleus from aws.greengrass.Nucleus-<version> to aws.greengrass.Nucleus-<version> but no component of type nucleus was included as target component

Greengrass nucleus に依存するコンポーネントをデプロイした場合に、コアデバイスが利用可能な最新のマイナーバージョンよりも前のバージョンの Greengrass nucleus を実行していると、このエラーが表示されることがあります。このエラーは、 AWS IoT Greengrass Core ソフトウェアがコンポーネントを最新の互換性のあるバージョンに自動的に更新しようとするために発生します。ただし、 AWS IoT Greengrass Core ソフトウェアは Greengrass nucleus が新しいマイナーバージョンに更新できないようにします。 AWSが提供するコンポーネントの一部は Greengrass nucleus の特定のマイナーバージョンに依存するためです。詳細については、「Greengrass nucleus の更新動作」を参照してください。

デプロイを改訂して、使用する Greengrass nucleus バージョンを指定する必要があります。次のいずれかを行います。

- デプロイを改訂して、コアデバイスが現在実行している Greengrass nucleus バージョンを指定します。
- デプロイを改訂して、Greengrass nucleus の新しいマイナーバージョンを指定します。このオプションを選択した場合は、Greengrass AWSnucleus の特定のマイナーバージョンに依存するが提供するすべてのコンポーネントのバージョンも更新する必要があります。詳細については、「AWS が提供したコンポーネント」を参照してください。

Error: com.aws.greengrass.deployment.exceptions.DeploymentException: Unable to process deployment. Greengrass launch directory is not set up or Greengrass is not set up as a system service

Greengrass のデバイスをあるグループから別のグループに移動し、Greengrass の再起動が必要なデプロイで元のグループに戻したときに、このエラーが表示されることがあります。

この問題を解決するには、デバイスの起動ディレクトリを再作成してください。また、Greengrass nucleus のバージョン 2.9.6 以降へのアップグレードを強く推奨します。

以下は、起動ディレクトリを再作成するための Linux スクリプトです。`fix_directory.sh` という名前のファイルにスクリプトを保存します。

```
#!/bin/bash

set -e

GG_ROOT=$1
GG_VERSION=$2

CURRENT="$GG_ROOT/alts/current"

if [ ! -L "$CURRENT" ]; then
  mkdir -p $GG_ROOT/alts/directory_fix
  echo "Relinking $GG_ROOT/alts/directory_fix to $CURRENT"
  ln -sf $GG_ROOT/alts/directory_fix $CURRENT
fi

TARGET=$(readlink $CURRENT)

if [[ ! -d "$TARGET" ]]; then
  echo "Creating directory: $TARGET"
  mkdir -p "$TARGET"
```

```
fi

DISTRO_LINK="$TARGET/distro"
DISTRO="$GG_ROOT/packages/artifacts-unarchived/aws.greengrass.Nucleus/$GG_VERSION/
aws.greengrass.nucleus/"
echo "Relinking Nucleus artifacts to $DISTRO_LINK"
ln -sf $DISTRO $DISTRO_LINK
```

スクリプトを実行するには、次のコマンドを実行します。

```
[root@ip-172-31-27-165 ~]# ./fix_directory.sh /greengrass/v2 2.9.5
Relinking /greengrass/v2/alts/directory_fix to /greengrass/v2/alts/current
Relinking Nucleus artifacts to /greengrass/v2/alts/directory_fix/distro
```

Info: com.aws.greengrass.deployment.exceptions.RetryableDeploymentDocumentDow Greengrass Cloud Service returned an error when getting full deployment configuration

このエラーは、コア デバイスが 7 KB (モノを対象とするデプロイの場合) または 31 KB (モノのグループを対象とするデプロイの場合) を超えるデプロイドキュメントである大規模なデプロイドキュメントを受信した場合に表示されることがあります。大規模なデプロイドキュメントを取得するには、コアデバイスの AWS IoT ポリシーで アクセス`greengrass:GetDeploymentConfiguration`許可を付与する必要があります。このエラーは、コアデバイスにこの権限がない場合に発生する可能性があります。このエラーが発生すると、デプロイは無期限に再試行され、そのステータスは [In progress] (進行中) (`IN_PROGRESS`) です。

この問題を解決するには、コアデバイスの AWS IoT ポリシーに アクセス`greengrass:GetDeploymentConfiguration`許可を追加します。詳細については、「コアデバイスの AWS IoT ポリシーを更新する」を参照してください。

Warn: com.aws.greengrass.deployment.DeploymentService: Failed to get thing group hierarchy

この警告は、コアデバイスがデプロイを受信し、コアデバイスの AWS IoT ポリシーで アクセス`greengrass:ListThingGroupsForCoreDevice`許可が許可されていない場合に表示されることがあります。デプロイを作成するとき、コアデバイスはこの権限を使用して、そのモノグループを識別し、コアデバイスを削除したすべてのモノグループのコンポーネントを削除します。コアデ

バイスが Greengrass nucleus v2.5.0 を実行している場合、デプロイは失敗します。コアデバイスが Greengrass nucleus v2.5.1 以降を実行している場合、デプロイは続行されますが、コンポーネントは削除されません。モノグループの削除動作の詳細については、「デバイスに AWS IoT Greengrass コンポーネントのデプロイ」を参照してください。

コアデバイスの動作を更新して、コアデバイスを削除するモノグループのコンポーネントを削除するには、コアデバイスの AWS IoT ポリシーに アクセス`greengrass:ListThingGroupsForCoreDevice`許可を追加します。詳細については、「コアデバイスの AWS IoT ポリシーを更新する」を参照してください。

Info: com.aws.greengrass.deployment.DeploymentDocumentDownloader: Calling Greengrass cloud to get full deployment configuration

コアデバイスは DEBUG ログレベルでエラーをログに記録するため、この情報メッセージがエラーなしで複数回出力される場合があります。この問題は、コアデバイスが大規模なデプロイドキュメントを受信した場合に、発生する可能性があります。この問題が発生すると、デプロイは無期限に再試行され、そのステータスは [In progress] (進行中) (`IN_PROGRESS`) です。この問題を解決する方法の詳細については、「こちらのトラブルシューティングエントリ」を参照してください。

Caused by: software.amazon.awssdk.services.greengrassv2data.model.GreengrassV2DataE null (Service: GreengrassV2Data, Status Code: 403, Request ID: <some_request_id>, Extended Request ID: null)

このエラーは、データプレーン API に アクセス`iot:Connect`許可がない場合に表示されることがあります。正しいポリシーがない場合は、 を受け取ります`GreengrassV2DataException: 403`。アクセス許可ポリシーを作成するには、 の手順に従いますAWS IoT ポリシーを作成する。

コアデバイスコンポーネントの問題

コアデバイスで Greengrass コンポーネントの問題のトラブルシューティングを行います。

トピック

- Warn: '<command>' is not recognized as an internal or external command
- Python スクリプトはメッセージを ログに記録しません
- デフォルト設定を変更してもコンポーネント設定が更新されません

- awsiot.greengrasscoreipc.model.UnauthorizedError
- com.aws.greengrass.authorization.exceptions.AuthorizationException: Duplicate policy ID "<id>" for principal "<componentList>"
- com.aws.greengrass.tes.CredentialRequestHandler: Error in retrieving AwsCredentials from TES (HTTP 400)
- com.aws.greengrass.tes.CredentialRequestHandler: Error in retrieving AwsCredentials from TES (HTTP 403)
- com.aws.greengrass.tes.CredentialsProviderError: Could not load credentials from any providers
- Received error when attempting to retrieve ECS metadata: Could not connect to the endpoint URL: "<tokenExchangeServiceEndpoint>"
- copyFrom: <configurationPath> is already a container, not a leaf
- com.aws.greengrass.componentmanager.plugins.docker.exceptions.DockerLoginException: Error logging into the registry using credentials - 'The stub received bad data.'
- java.io.IOException: Cannot run program "cmd" ...: [LogonUser] The password for this account has expired.
- aws.greengrass.StreamManager: Instant exceeds minimum or maximum instant

Warn: '<command>' is not recognized as an internal or external command

AWS IoT Greengrass Core ソフトウェアがコンポーネントのライフサイクルスクリプトでコマンドを実行できない場合、Greengrass コンポーネントのログにこのエラーが表示されることがあります。このエラーの結果、コンポーネントの状態は `BROKEN` になります。このエラーは、`ggc_user` などのコンポーネントを実行するシステムユーザーが PATH 内のフォルダでコマンドの実行可能ファイルを見つけられない場合に発生する可能性があります。

Windows デバイスでは、実行可能ファイルを含むフォルダが、コンポーネントを実行するシステムユーザーの `PATH` にあることを確認します。`PATH` から欠落している場合は、次のいずれかを実行します。

- 実行可能ファイルのフォルダを、すべてのユーザーを利用できる `PATH` システム変数に追加します。次に、コンポーネントを再起動します。

  Greengrass nucleus 2.5.0 を実行する場合は、 `PATH`システム変数を更新した後、 AWS IoT Greengrass Core ソフトウェアを再起動して、更新された でコンポーネントを実行する必要がありますPATH。ソフトウェアを再起動PATHした後に AWS IoT Greengrass Core ソフトウェアが

更新された を使用しない場合は、デバイスを再起動して再試行してください。詳細については、「AWS IoT Greengrass Core ソフトウェアを実行する」を参照してください。

- コンポーネントを実行するシステムユーザーの PATH ユーザー変数に実行可能ファイルのフォルダを追加します。

Python スクリプトはメッセージをログに記録しません

Greengrass コアデバイスは、コンポーネントの問題を特定するために使用できるログを収集します。Python スクリプトの stdout と stderr のメッセージがコンポーネントログに表示されない場合、Python でこれらの標準出力ストリームのバッファをフラッシュするか、バッファリングを無効にする必要がある場合があります。次のいずれかを実行します。

- -u 引数を指定してPythonを実行し、stdout と stderr のバッファリングを無効にします。

  Linux or Unix

  ```
  python3 -u hello_world.py
  ```

  Windows

  ```
  py -3 -u hello_world.py
  ```

- コンポーネントの recipe で Setenv を使用して、PYTHONUNBUFFERED 環境変数を空でない文字列に設定します。この環境変数は stdout と stderr のバッファリングを無効にします。
- stdout または stderr Streams のバッファをフラッシュします。次のいずれかを行います。
  - 印刷時にメッセージをフラッシュします。

  ```
  import sys

  print('Hello, error!', file=sys.stderr, flush=True)
  ```

  - 印刷後にメッセージをフラッシュします。ストリームをフラッシュする前に、複数のメッセージを送信できます。

  ```
  import sys

  print('Hello, error!', file=sys.stderr)
  sys.stderr.flush()
  ```

Python スクリプトがログメッセージを出力することを確認する方法の詳細については、「AWS IoT Greengrass ログのモニタリング」を参照してください。

デフォルト設定を変更してもコンポーネント設定が更新されません

コンポーネントの recipe で `DefaultConfiguration` を変更した場合、デプロイ中に新しいデフォルト設定がコンポーネントの既存の設定を置き換えることはありません。新しいデフォルト設定を適用するには、コンポーネントの設定をデフォルト設定にリセットする必要があります。コンポーネントをデプロイする場合には、更新のリセットとして空の文字列を 1 つ指定します。

Console

パスのリセット

```
[""]
```

AWS CLI

次のコマンドは、コアデバイスにデプロイを作成します。

```
aws greengrassv2 create-deployment --cli-input-json file://reset-configuration-
deployment.json
```

`reset-configuration-deployment.json` ファイルには、次の JSON ドキュメントが含まれています。

```
{
  "targetArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
  "deploymentName": "Deployment for MyGreengrassCore",
  "components": {
    "com.example.HelloWorld": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {,
        "reset": [""]
      }
    }
  }
}
```

Greengrass CLI

次の Greengrass CLI コマンドは、コアデバイスにローカルデプロイを作成します。

```
sudo greengrass-cli deployment create \
  --recipeDir recipes \
  --artifactDir artifacts \
  --merge "com.example.HelloWorld=1.0.0" \
  --update-config reset-configuration-deployment.json
```

`reset-configuration-deployment.json` ファイルには、次の JSON ドキュメントが含まれています。

```
{
  "com.example.HelloWorld": {
    "RESET": [""]
  }
}
```

awsiot.greengrasscoreipc.model.UnauthorizedError

コンポーネントにリソースに対する IPC オペレーションを実行するアクセス許可がない場合に、Greengrass コンポーネントのログにこのエラーが表示されることがあります。コンポーネントに IPC オペレーションを呼び出すアクセス許可を付与するには、コンポーネントの設定で IPC 承認ポリシーを定義します。詳細については、「コンポーネントに IPC オペレーションの実行を許可する」を参照してください。

> **Tip**
>
> コンポーネントの recipe で `DefaultConfiguration` を変更した場合は、コンポーネントの設定を新しいデフォルト設定にリセットする必要があります。コンポーネントをデプロイする場合には、更新のリセットとして空の文字列を 1 つ指定します。詳細については、「デフォルト設定を変更してもコンポーネント設定が更新されません」を参照してください。

com.aws.greengrass.authorization.exceptions.AuthorizationException: Duplicate policy ID "<id>" for principal "<componentList>"

このエラーは、コアデバイスのすべてのコンポーネントを含め、複数の IPC 承認ポリシーが同じポリシー ID を使用している場合に表示されることがあります。

コンポーネントの IPC 承認ポリシーを確認し、重複があれば修正して、再試行します。一意のポリシー ID を作成するには、コンポーネント名、IPC サービス名、および順番号を組み合わせることをお勧めします。詳細については、「コンポーネントに IPC オペレーションの実行を許可する」を参照してください。

> **Tip**
>
> コンポーネントの recipe で `DefaultConfiguration` を変更した場合は、コンポーネントの設定を新しいデフォルト設定にリセットする必要があります。コンポーネントをデプロイする場合には、更新のリセットとして空の文字列を 1 つ指定します。詳細については、「デフォルト設定を変更してもコンポーネント設定が更新されません」を参照してください。

com.aws.greengrass.tes.CredentialRequestHandler: Error in retrieving AwsCredentials from TES (HTTP 400)

このエラーは、コアデバイスがトークン交換サービス から AWS 認証情報を取得できない場合に表示されることがあります。HTTP 400 ステータスコードは、コアデバイスのトークン交換 IAM ロールが存在しないか、 AWS IoT 認証情報プロバイダーが引き受けることを許可する信頼関係がないためにこのエラーが発生したことを示します。

以下の操作を実行します。

1. コアデバイスが使用するトークン交換ロールを特定します。エラーメッセージには、コアデバイスの AWS IoT ロールエイリアスが含まれており、トークン交換ロールを指します。開発用コンピュータで次のコマンドを実行し、 をエラーメッセージの AWS IoT ロールエイリアスの名前*MyGreengrassCoreTokenExchangeRoleAlias*に置き換えます。

   ```
   aws iot describe-role-alias --role-alias MyGreengrassCoreTokenExchangeRoleAlias
   ```

   レスポンスには、トークン交換 IAM ロールの Amazon リソースネーム (ARN) が含まれます。

```
{
  "roleAliasDescription": {
    "roleAlias": "MyGreengrassCoreTokenExchangeRoleAlias",
    "roleAliasArn": "arn:aws:iot:us-west-2:123456789012:rolealias/
MyGreengrassCoreTokenExchangeRoleAlias",
    "roleArn": "arn:aws:iam::123456789012:role/MyGreengrassV2TokenExchangeRole",
    "owner": "123456789012",
    "credentialDurationSeconds": 3600,
    "creationDate": "2021-02-05T16:46:18.042000-08:00",
    "lastModifiedDate": "2021-02-05T16:46:18.042000-08:00"
  }
}
```

2. ロールが存在することを確認します。次のコマンドを実行し、*MyGreengrassV2TokenExchangeRole* をトークン交換ロールの名前に置き換えます。

```
aws iam get-role --role-name MyGreengrassV2TokenExchangeRole
```

コマンドが NoSuchEntity エラーを返した場合、ロールは存在しないため、作成する必要があります。このロールを作成および設定する方法の詳細については、「コアデバイスが AWS サービスを操作できるように認証する」を参照してください。

3. ロールに、 AWS IoT 認証情報プロバイダーが引き受けることを許可する信頼関係があることを確認します。前のステップのレスポンスには AssumeRolePolicyDocument が含まれ、これはロールの信頼関係を定義します。ロールは、credentials.iot.amazonaws.com が引き受けることを許可する信頼関係を定義する必要があります。このドキュメントは次の例のようになります。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.iot.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

ロールの信頼関係で `credentials.iot.amazonaws.com` が引き受けることが許可されていない場合、ロールにこの信頼関係を追加する必要があります。詳細については、「AWS Identity and Access Management ユーザーガイド」の「ロールの修正」を参照してください。

com.aws.greengrass.tes.CredentialRequestHandler: Error in retrieving AwsCredentials from TES (HTTP 403)

このエラーは、コアデバイスがトークン交換サービス から AWS 認証情報を取得できない場合に表示されることがあります。HTTP 403 ステータスコードは、コアデバイスの AWS IoT ポリシーがコアデバイスの AWS IoT ロールエイリアスに アクセス`iot:AssumeRoleWithCertificate`許可を付与していないためにこのエラーが発生したことを示します。

コアデバイスの AWS IoT ポリシーを確認し、コアデバイスの AWS IoT ロールエイリアスの アクセス`iot:AssumeRoleWithCertificate`許可を追加します。エラーメッセージには、コアデバイスの現在の AWS IoT ロールエイリアスが含まれます。このアクセス許可とコアデバイスの AWS IoT ポリシーを更新する方法の詳細については、AWS IoT Greengrass V2 コアデバイス向けの最低限の AWS IoT ポリシー「」および「」を参照してくださいコアデバイスの AWS IoT ポリシーを更新する。

com.aws.greengrass.tes.CredentialsProviderError: Could not load credentials from any providers

このエラーは、コンポーネントが AWS 認証情報をリクエストしようとして、トークン交換サービスに接続できない場合に表示されることがあります。

以下の操作を実行します。

- コンポーネントがトークン交換サービスコンポーネント `aws.greengrass.TokenExchangeService` への依存関係を宣言していることを確認します。宣言していない場合は、依存関係を追加して、コンポーネントを再デプロイします。
- コンポーネントが Docker で実行される場合は、Docker コンテナコンポーネント (Linux) でAWS 認証情報の使用 に従って適切なネットワーク設定と環境変数を適用するようにしてください。
- コンポーネントが NodeJS で記述されている場合は、dns.setDefaultResultOrder を に設定しますipv4first。

- `::1` で始まり、`localhost` を含むエントリがないか `/etc/hosts` を調べます。エントリを削除して、そのエントリがコンポーネントが間違ったアドレスでトークン交換サービスに接続した原因であったかどうかを確認します。

Received error when attempting to retrieve ECS metadata: Could not connect to the endpoint URL: "<tokenExchangeServiceEndpoint>"

このエラーは、コンポーネントがトークン交換サービスを実行せず、コンポーネントが AWS 認証情報をリクエストしようとした場合に表示されることがあります。

以下の操作を実行します。

- コンポーネントがトークン交換サービスコンポーネント `aws.greengrass.TokenExchangeService` への依存関係を宣言していることを確認します。宣言していない場合は、依存関係を追加して、コンポーネントを再デプロイします。
- コンポーネントが `install` lifecycle. AWS IoT Greengrass doesn で AWS 認証情報を使用しているかどうかを確認し、`install`ライフサイクル中にトークン交換サービスの可用性を保証しません。コンポーネントを更新して、 AWS 認証情報を使用するコードを `startup` または `run` ライフサイクルに移動してからコンポーネントを再デプロイします。

copyFrom: <configurationPath> is already a container, not a leaf

このエラーは、設定値をコンテナタイプ (リストまたはオブジェクト) から非コンテナタイプ (文字列、数値、またはブール値) に変更したときに表示されることがあります。以下の操作を実行します。

1. コンポーネントの recipe をチェックして、デフォルト設定で、その設定値がリストまたはオブジェクトに設定されているかどうかを確認します。その場合は、その設定値を削除または変更します。
2. その設定値をデフォルト値にリセットするデプロイを作成します。詳細については、デプロイの作成およびコンポーネント設定の更新を参照してください。

その後で、その設定値を文字列、数値、またはブール値に設定できます。

com.aws.greengrass.componentmanager.plugins.docker.exceptions.DockerLoginException: Error logging into the registry using credentials - 'The stub received bad data.'

[Docker application manager component] (Docker アプリケーションマネージャーコンポーネント) が Amazon Elastic Container Registry (Amazon ECR) のプライベートリポジトリから Docker イメージをダウンロードしようとすると、Greengrass nucleus ログにこのエラーが表示される場合があります。このエラーは、`wincred` Docker credential helper (Docker 認証情報ヘルパー)(`docker-credential-wincred`) を使用した場合に発生します。その結果、Amazon ECR はログイン認証情報を保存できません。

次のいずれかのアクションを実行します。

- `wincred` Docker 認証情報ヘルパーを使用しない場合、コアデバイスから `docker-credential-wincred` プログラムを削除してください。
- `wincred` Docker 認証情報ヘルパーを使用する場合、以下の操作を行います。
  1. コアデバイスの `docker-credential-wincred` プログラムの名前を変更します。`wincred` を Windows Docker 認証情報ヘルパーの新しい名前に置き換えます。例えば、名前を `docker-credential-wincredreal` に変更できます。
  2. Docker 設定 (`.docker/config.json`) の `credsStore` オプションを更新し、Windows Docker 認証情報ヘルパーの新しい名前を使用します。例えば、プログラムの名前を `docker-credential-wincredreal` に変更した場合は、`credsStore` オプションを `wincredreal` に更新します。

     ```
     {
       "credsStore": "wincredreal"
     }
     ```

java.io.IOException: Cannot run program "cmd" ...: [LogonUser] The password for this account has expired.

このエラーは、`ggc_user` など、コンポーネントのプロセスを実行するシステムユーザーのパスワードの有効期限が切れている場合に、Windows コアデバイスに表示されることがあります。その結果、 AWS IoT Greengrass Core ソフトウェアはそのシステムユーザーとしてコンポーネントプロセスを実行できません。

Greengrass システムユーザーのパスワードを更新するには

1. 管理者として次のコマンドを実行して、ユーザーのパスワードを設定します。*ggc_user* をシステムユーザーに置き換え、*password* を設定するパスワードに置き換えます。

   ```
   net user ggc_user password
   ```

2. PsExec ユーティリティを使用して、ユーザーの新しいパスワードを LocalSystem アカウントの認証情報マネージャーインスタンスに保存します。*password* を設定したユーザーのパスワードに置き換えます。

   ```
   psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
   ```

> **Tip**
>
> Windows の構成によっては、ユーザーのパスワードの期限切れが、将来の日付に設定されている場合があります。Greengrass アプリケーションの動作を継続させるためには、パスワードの有効期限を追跡し、その期限が切れる前に更新します。ユーザーのパスワードには、期限切れを起こさないような設定も可能です。
>
> - ユーザーとパスワードの有効期限を確認するには、次のコマンドを実行します。
>
>   ```
>   net user ggc_user | findstr /C:expires
>   ```
>
> - ユーザーのパスワードが期限切れにならないように設定するには、次のコマンドを実行します。
>
>   ```
>   wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
>   ```
>
> - wmic コマンドが廃止された Windows 10 以降を使用している場合は、次の PowerShell コマンドを実行します。
>
>   ```
>   Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
>    'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
>   ```

aws.greengrass.StreamManager: Instant exceeds minimum or maximum instant

ストリームマネージャ v2.0.7 を v2.0.8 と v2.0.11 の間のバージョンにアップグレードする場合、コンポーネントの起動に失敗すると、ストリームマネージャーコンポーネントのログに次のエラーが表示される場合があります。

```
2021-07-16T00:54:58.568Z [INFO] (Copier) aws.greengrass.StreamManager:
 stdout. Caused by: com.fasterxml.jackson.databind.JsonMappingException:
 Instant exceeds minimum or maximum instant (through reference chain:
 com.amazonaws.iot.greengrass.streammanager.export.PersistedSuccessExportStatesV1["lastExportTi
 {scriptName=services.aws.greengrass.StreamManager.lifecycle.startup.script,
 serviceName=aws.greengrass.StreamManager, currentState=STARTING}
2021-07-16T00:54:58.579Z [INFO] (Copier) aws.greengrass.StreamManager: stdout.
 Caused by: java.time.DateTimeException: Instant exceeds minimum or maximum instant.
 {scriptName=services.aws.greengrass.StreamManager.lifecycle.startup.script,
 serviceName=aws.greengrass.StreamManager, currentState=STARTING}
```

ストリームマネージャー v2.0.7 をデプロイし、それ以降のバージョンにアップグレードする場合は、ストリームマネージャー v2.0.12 に直接アップグレードする必要があります。ストリームマネージャコンポーネントの詳細については、「ストリームマネージャー」を参照してください。

コアデバイスの Lambda 関数コンポーネントの問題

コアデバイスでの Lambda 関数コンポーネントの問題のトラブルシューティングを行います。

トピック

- The following cgroup subsystems are not mounted: devices, memory
- ipc_client.py:64,HTTP Error 400:Bad Request, b'No subscription exists for the source <label-or-lambda-arn> and subject <label-or-lambda-arn>

The following cgroup subsystems are not mounted: devices, memory

次の場合に、コンテナ化した Lambda 関数を実行すると、このエラーが表示されることがあります。

- コアデバイスで、メモリまたはデバイス cgroup に対して cgroup v1 が有効になっていない。

- コアデバイスで cgroup v2 が有効になっている。Greengrass Lambda 関数は cgroup v1 を必要とし、cgroup v1 と v2 は相互に排他的です。

cgroups v1 を有効にするには、次の Linux カーネルパラメータを使用してデバイスを起動します。

```
cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
```

> **Tip**
>
> Raspberry Pi で、`/boot/cmdline.txt` ファイルを編集して、デバイスのカーネルパラメータを設定します。

ipc_client.py:64,HTTP Error 400:Bad Request, b'No subscription exists for the source <label-or-lambda-arn> and subject <label-or-lambda-arn>

このエラーは、レガシーサブスクリプションルーターコンポーネント でサブスクリプションを指定せずに、 AWS IoT Greengrass Core SDK を使用する V1 Lambda 関数を V2 コアデバイスで実行すると表示されることがあります。この問題を解決するには、レガシーサブスクリプションルーターをデプロイして設定し、必要なサブスクリプションを指定します。詳細については、「V1 Lambda 関数をインポートする」を参照してください。

コンポーネントのバージョンが廃止された

コアデバイスのコンポーネントのバージョンが廃止されているとき、Personal Health Dashboard (PHD) に通知が表示される場合があります。そのコンポーネントのバージョンは、廃止されてから 60 分以内にこの通知を PHD に送信します。

どのデプロイを修正する必要があるかを確認するには、 AWS Command Line Interface を使用して次の操作を行います。

1. 次のコマンドを実行し、コアデバイスのリストを取得します。

   ```
   aws greengrassv2 list-core-devices
   ```

2. 次のコマンドを実行し、手順 1 の各コアデバイス上のコンポーネントのステータスを取得します。*coreDeviceName* をクエリする各コアデバイスの名前に置き換えます。

```
aws greengrassv2 list-installed-components --core-device-thing-name coreDeviceName
```

3. 前の手順でインストールした、廃止されたコンポーネントのバージョンのコアデバイスを収集します。
4. 次のコマンドを実行し、手順 3 の各コアデバイスに対するすべてのデプロイジョブのステータスを取得します。*coreDeviceName* をクエリするコアデバイスの名前に置き換えます。

   ```
   aws greengrassv2 list-effective-deployments --core-device-thing-name coreDeviceName
   ```

   レスポンスには、コアデバイスのデプロイジョブのリストが含まれます。デプロイを修正して別のコンポーネントのバージョンを選択できます。デプロイを修正する方法の詳細については、「デプロイの修正」を参照してください。

Greengrass コマンドラインインターフェイスの問題

Greengrass CLI の問題のトラブルシューティング。

トピック

- java.lang.RuntimeException: Unable to create ipc client

java.lang.RuntimeException: Unable to create ipc client

このエラーは、Greengrass CLI コマンドを実行し、 AWS IoT Greengrass Core ソフトウェアがインストールされている場所とは異なるルートフォルダを指定すると表示されることがあります。

次のいずれかを実行してルートパスを設定し、 を AWS IoT Greengrass Core ソフトウェアのインストールパス*/greengrass/v2*に置き換えます。

- GGC_ROOT_PATH 環境変数を */greengrass/v2* に設定します。
- 次の例のように、コマンドに --ggcRootPath */greengrass/v2* 引数を追加します。

```
greengrass-cli --ggcRootPath /greengrass/v2 <command> <subcommand> [arguments]
```

AWS Command Line Interface 問題

AWS CLI の問題をトラブルシューティングします AWS IoT Greengrass V2。

トピック

Error: Invalid choice: 'greengrassv2'

このエラーは、 AWS CLI （ 例:) を使用して AWS IoT Greengrass V2 コマンドを実行すると表示されることがありますaws greengrassv2 list-core-devices。

このエラーは、 をサポート AWS CLI していないバージョンの があることを示しています AWS IoT Greengrass V2。 AWS IoT Greengrass V2 で を使用するには AWS CLI、次のいずれかのバージョン以降が必要です。

- 最小 AWS CLI V1 バージョン: v1.18.197
- 最小 AWS CLI V2 バージョン: v2.1.11

> **Tip**
>
> 次のコマンドを実行して、 AWS CLI 使用している のバージョンを確認できます。
>
> ```
> aws --version
> ```

この問題を解決するには、 を をサポートする新しいバージョン AWS CLI に更新します AWS IoT Greengrass V2。詳細については、「AWS Command Line Interface ユーザーガイド」の「AWS CLI のインストール、更新、およびアンインストール」を参照してください。

詳細なデプロイエラーコード

これらのセクションのエラーコードと解決方法を参考にして、Greengrass nucleus バージョン 2.8.0 以降を使用する際のコンポーネントのデプロイの問題の解決に役立ててください。

Greengrass nucleus は、入手可能な最も具体的でないコードから最も具体的なコードまでの階層として、デプロイエラーを報告します。この階層を使用すると、デプロイエラーの原因を特定するのに役立ちます。例えば、エラー階層としては、次のようなものが考えられます。

- DEPLOYMENT_FAILURE
  - ARTIFACT_DOWNLOAD_ERROR
    - IO_ERROR
      - DISK_SPACE_CRITICAL

エラーコードは、複数のタイプに整理されています。各タイプは、発生する可能性があるエラーのクラスを表します。AWS IoT Greengrass は、コンソール、API、および AWS CLI でこれらのエラータイプを報告します。エラー階層で報告されるエラーによっては、複数のエラータイプが存在する可能性があります。前述の例では、返されるエラータイプは `DEVICE_ERROR` です。

タイプは次のとおりです。

- PERMISSION_ERROR — 許可を必要とするオペレーションへのアクセスが拒否されました。
- REQUEST_ERROR — デプロイドキュメント内の問題が原因でエラーが発生しました。
- COMPONENT_RECIPE_ERROR — コンポーネント recipe の問題が原因でエラーが発生しました。
- AWS_COMPONENT_ERROR – AWS が提供するコンポーネントを開始または削除するときにエラーが発生しました。
- USER_COMPONENT_ERROR — ユーザーコンポーネントの起動時または削除時にエラーが発生しました。
- COMPONENT_ERROR – コンポーネントの開始時または削除時にエラーが発生しましたが、Greengrass nucleus は、そのコンポーネントが、AWS が提供するコンポーネントであるか、またはユーザーコンポーネントであるかを判断できませんでした。
- DEVICE_ERROR — ローカル I/O でエラーが発生したか、別のデバイスエラーが発生しました。
- DEPENDENCY_ERROR — デプロイが Amazon S3 からアーティファクトをダウンロードしたり、ECR レジストリからイメージをプルしたりできませんでした。
- HTTP_ERROR — HTTP リクエストでエラーが発生しました。
- NETWORK_ERROR — デバイスネットワークでエラーが発生しました。
- NUCLEUS_ERROR — Greengrass nucleus がコンポーネントまたはアクティブな nucleus バージョンを見つけることができませんでした。

- SERVER_ERROR — リクエストに対するレスポンスとして、サーバーが 500 エラーを返しました。
- CLOUD_SERVICE_ERROR – AWS IoT Greengrass クラウドサービスでエラーが発生しました。
- UNKNOWN_ERROR — コンポーネントによって未チェックの例外がスローされました。

このセクションのエラーの多くは、AWS IoT Greengrass Core ログで追加情報を報告します。これらのログは、コアデバイスのローカルファイルシステムに保存されます。AWS IoT Greengrass Core コアソフトウェアと個別の各コンポーネントのログがあります。ログへのアクセスに関する詳細については、「ファイル システム ログをアクセス」を参照してください。

アクセス許可エラー

ACCESS_DENIED

このエラーは、許可が正しく設定されていないために、AWS サービスオペレーションが 403 エラーを返した場合に表示されることがあります。詳細については、より具体的なエラーコードを確認してください。

GET_DEPLOYMENT_CONFIGURATION_ACCESS_DENIED

このエラーは、AWS IoT ポリシーで `GetDeploymentConfiguration` オペレーションを呼び出すための許可が付与されていない場合に表示されることがあります。`greengrass::GetDeploymentConfiguration` 許可をコアデバイスのポリシーに追加します。

GET_COMPONENT_VERSION_ARTIFACT_ACCESS_DENIED

このエラーは、コアデバイスの AWS IoT ポリシーで `greengrass:GetComponentVersionArtifact` 許可が付与されていない場合に表示されることがあります。許可をコアデバイスのポリシーに追加します。

RESOLVE_COMPONENT_CANDIDATES_ACCESS_DENIED

このエラーは、コアデバイスの AWS IoT ポリシーで `greengrass:ResolveComponentCandidates` 許可が付与されていない場合に表示されることがあります。許可をコアデバイスのポリシーに追加します。

GET_ECR_CREDENTIAL_ERROR

このエラーは、デプロイが ECR のプライベートレジストリで認証できなかった場合に表示されることがあります。特定のエラーがないかを口グで確認してから、デプロイを再試行してください。

USER_NOT_AUTHORIZED_FOR_DOCKER

このエラーは、Greengrass ユーザーに Docker を使用する権限がない場合に表示されることがあります。Greengrass をルートとして実行していること、またはユーザーが `docker` グループに追加されていることを確認してください。その後、デプロイを再試行してください。

S3_ACCESS_DENIED

このエラーは、Amazon S3 オペレーションが 403 エラーを返した場合に表示されることがあります。詳細については、その他のエラーコードまたはログを確認してください。

S3_HEAD_OBJECT_ACCESS_DENIED

このエラーは、AWS IoT Greengrass Core ソフトウェアがコンポーネントの recipe で指定した S3 オブジェクト URL からコンポーネントアーティファクトをダウンロードすることをデバイスのトークン交換ロールが許可しない場合、またはコンポーネントアーティファクトが利用できない場合に表示されることがあります。トークン交換ロールで、アーティファクトが利用可能で、かつ、アーティファクトが存在している S3 オブジェクト URL の `s3:GetObject` が許可されていることを確認してください。

S3_GET_BUCKET_LOCATION_ACCESS_DENIED

このエラーは、アーティファクトが利用可能な Amazon S3 バケットに対して、デバイスのトークン交換ロールが `s3:GetBucketLocation` 許可を付与していない場合に表示されることがあります。デバイスが許可を付与していることを確認してから、デプロイを再試行してください。

S3_GET_OBJECT_ACCESS_DENIED

このエラーは、AWS IoT Greengrass Core ソフトウェアがコンポーネントの recipe で指定した S3 オブジェクト URL からコンポーネントアーティファクトをダウンロードすることをデバイスのトークン交換ロールが許可しない場合、またはコンポーネントアーティファクトが利用できない場合に表示されることがあります。トークン交換ロールで、アーティファクトが利用可能で、かつ、アーティファクトが存在している S3 オブジェクト URL の `s3:GetObject` が許可されていることを確認してください。

リクエストエラー

NUCLEUS_MISSING_REQUIRED_CAPABILITIES

このエラーは、デプロイの nucleus バージョンが、リクエストされたオペレーション (大規模な設定のダウンロードや Linux リソース制限の設定など) に対応していない場合に表示されること

があります。オペレーションをサポートする nucleus バージョンでデプロイを再試行してください。

MULTIPLE_NUCLEUS_RESOLVED_ERROR

このエラーは、デプロイが複数の nucleus コンポーネントをデプロイしようとした場合に表示されることがあります。ログでエラーの原因を確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

COMPONENT_CIRCULAR_DEPENDENCY_ERROR

このエラーは、デプロイ内の 2 つのコンポーネントが相互に依拠している場合に表示されることがあります。デプロイ内のコンポーネントが相互に依拠しないように、コンポーネントの設定を見直します。

UNAUTHORIZED_NUCLEUS_MINOR_VERSION_UPDATE

このエラーは、デプロイ内のコンポーネントが nucleus のマイナーバージョン更新を必要とするものの、そのバージョンがデプロイで指定されていない場合に表示されることがあります。これは、別のバージョンに依拠するコンポーネントが、意図せずにマイナーバージョン更新されるのを防ぐのに役立ちます。デプロイに新しいマイナー nucleus バージョンを含めてください。

MISSING_DOCKER_APPLICATION_MANAGER

このエラーは、Docker アプリケーションマネージャーをデプロイせずに Docker コンポーネントをデプロイする場合に表示されることがあります。デプロイに Docker アプリケーションマネージャーが含まれていることを確認してください。

MISSING_TOKEN_EXCHANGE_SERVICE

このエラーは、デプロイがトークン交換サービスをデプロイせずに、プライベート ECR レジストリから Docker イメージアーティファクトをダウンロードする場合に表示されることがあります。デプロイにトークン交換サービスが含まれていることを確認してください。

COMPONENT_VERSION_REQUIREMENTS_NOT_MET

このエラーは、バージョン制約が競合しているか、コンポーネントバージョンが存在しない場合に表示されることがあります。詳細については、「Error: com.aws.greengrass.componentmanager.exceptions.NoAvailableComponentVersionException: Failed to negotiate component <name> version with cloud and no local applicable version satisfying requirement <requirements>」を参照してください。

THROTTLING_ERROR

このエラーは、AWS サービスオペレーションがレートクォータを超えた場合に表示されることがあります。デプロイを再試行します。

CONFLICTED_REQUEST

このエラーは、デプロイで一度に複数のオペレーションを実行しようとしているために、AWS サービスオペレーションが 409 エラーを返した場合に表示されることがあります。デプロイを再試行します。

RESOURCE_NOT_FOUND

このエラーは、リソースが見つからなかったために AWS サービスオペレーションが 404 エラーを返した場合に表示されることがあります。見つからないリソースについては、ログを確認してください。

RUN_WITH_CONFIG_NOT_VALID

このエラーは、コンポーネントを実行するために指定された `posixUser`、`posixGroup`、または `windowsUser` 情報が有効でない場合に表示されることがあります。ユーザーが有効であることを確認してから、デプロイを再試行してください。

UNSUPPORTED_REGION

このエラーは、デプロイに指定されたリージョンが AWS IoT Greengrass によってサポートされていない場合に表示されることがあります。リージョンを確認して、デプロイを再試行してください。

IOT_CRED_ENDPOINT_NOT_VALID

このエラーは、設定で指定された AWS IoT 認証情報エンドポイントが有効でない場合に表示されることがあります。エンドポイントを確認して、リクエストを再試行してください。

IOT_DATA_ENDPOINT_NOT_VALID

このエラーは、設定で指定された AWS IoT データエンドポイントが有効でない場合に表示されることがあります。エンドポイントを確認して、リクエストを再試行してください。

S3_HEAD_OBJECT_RESOURCE_NOT_FOUND

このエラーは、コンポーネントのアーティファクトが、コンポーネントの recipe で指定した S3 オブジェクト URL では利用できない場合に表示されることがあります。アーティファクトを S3 バケットにアップロードしたこと、アーティファクトの URI がバケット内のアーティファクトの S3 オブジェクト URL と一致していることを確認します。

S3_GET_BUCKET_LOCATION_RESOURCE_NOT_FOUND

このエラーは、Amazon S3 バケットが見つからない場合に表示されることがあります。バケットが存在することを確認して、デプロイを再試行してください。

S3_GET_OBJECT_RESOURCE_NOT_FOUND

このエラーは、コンポーネントのアーティファクトが、コンポーネントの recipe で指定した S3 オブジェクト URL では利用できない場合に表示されることがあります。アーティファクトを S3 バケットにアップロードしたこと、アーティファクトの URI がバケット内のアーティファクトの S3 オブジェクト URL と一致していることを確認します。

IO_MAPPING_ERROR

このエラーは、デプロイドキュメントまたは recipe を解析しているときに I/O エラーが発生した場合に表示されることがあります。詳細については、その他のエラーコードまたはログを確認してください。

コンポーネント recipe エラー

RECIPE_PARSE_ERROR

このエラーは、recipe の構造にエラーがあるためにデプロイ recipe を解析できなかった場合に表示されることがあります。recipe が正しくフォーマットされていることを確認して、デプロイを再試行してください。

RECIPE_METADATA_PARSE_ERROR

このエラーは、クラウドからダウンロードしたデプロイ recipe のメタデータを解析できなかった場合に表示されることがあります。に連絡しますAWS Support

ARTIFACT_URI_NOT_VALID

このエラーは、recipe 内のアーティファクト URI が正しくフォーマットされていない場合に表示されることがあります。ログで有効でない URI を確認し、recipe 内の URI を更新してから、デプロイを再試行してください。

S3_ARTIFACT_URI_NOT_VALID

このエラーは、recipe 内のアーティファクトの Amazon S3 URI が有効でない場合に表示されることがあります。ログで有効でない URI を確認し、recipe 内の URI を更新してから、デプロイを再試行してください。

DOCKER_ARTIFACT_URI_NOT_VALID

このエラーは、recipe 内のアーティファクトの Docker URI が有効でない場合に表示されることがあります。ログで有効でない URI を確認し、recipe 内の URI を更新してから、デプロイを再試行してください。

EMPTY_ARTIFACT_URI

このエラーは、アーティファクトの URI が recipe で指定されていない場合に表示されることがあります。URI が欠落しているアーティファクトをログで確認し、recipe 内の URI を更新してから、デプロイを再試行してください。

EMPTY_ARTIFACT_SCHEME

このエラーは、アーティファクトの URI スキームが定義されていない場合に表示されることがあります。ログで有効でない URI を確認し、recipe 内の URI を更新してから、デプロイを再試行してください。

UNSUPPORTED_ARTIFACT_SCHEME

このエラーは、実行中の nucleus バージョンによって URI スキームがサポートされていない場合に表示されることがあります。URI が有効でないか、nucleus のバージョンを更新する必要があります。URI が有効でない場合は、ログで有効でない URI を確認し、recipe 内の URI を更新してから、デプロイを再試行してください。

RECIPE_MISSING_MANIFEST

このエラーは、マニフェストセクションが recipe に含まれていない場合に表示されることがあります。マニフェストを recipe に追加して、デプロイを再試行してください。

RECIPE_MISSING_ARTIFACT_HASH_ALGORITHM

このエラーは、ローカルではないアーティファクトがハッシュアルゴリズムのない recipe 内で指定された場合に表示されることがあります。アーティファクトにアルゴリズムを追加してから、リクエストを再試行してください。

ARTIFACT_CHECKSUM_MISMATCH

このエラーは、ダウンロードしたアーティファクトのダイジェストが recipe で指定されたものと異なる場合に表示されることがあります。recipe に正しいダイジェストが含まれていることを確認してから、デプロイを再試行してください。詳細については、「Error: com.aws.greengrass.componentmanager.exceptions.ArtifactChecksumMismatchException: Integrity check for downloaded artifact failed. Probably due to file corruption.」を参照してください。

COMPONENT_DEPENDENCY_NOT_VALID

このエラーは、デプロイ recipe で指定された依存関係タイプが有効でない場合に表示されることがあります。recipe を確認してから、リクエストを再試行してください。

CONFIG_INTERPOLATE_ERROR

このエラーは、recipe 変数を補間する場合に表示されることがあります。詳細については、ログを確認してください。

IO_MAPPING_ERROR

このエラーは、デプロイドキュメントまたは recipe を解析しているときに I/O エラーが発生した場合に表示されることがあります。詳細については、その他のエラーコードまたはログを確認してください。

AWS コンポーネントエラー、ユーザーコンポーネントエラー、コンポーネントエラー

コンポーネントに問題があると、次のエラーコードが返されます。実際に報告されるエラータイプは、エラーを発生させた特定のコンポーネントによって異なります。Greengrass nucleus がコンポーネントを AWS IoT Greengrass によって提供されたものとして識別した場合、`AWS_COMPONENT_ERROR` を返します。コンポーネントがユーザーコンポーネントとして識別された場合、Greengrass nucleus は `USER_COMPONENT_ERROR` を返します。Greengrass nucleus が判別できない場合は、`COMPONENT_ERROR` を返します。

COMPONENT_UPDATE_ERROR

このエラーは、デプロイ中にコンポーネントが更新されない場合に表示されることがあります。その他のエラーコードを確認するか、ログでエラーの原因を確認してください。

COMPONENT_BROKEN

このエラーは、デプロイ中にコンポーネントが壊れた場合に表示されることがあります。コンポーネントログでエラーの詳細を確認してから、デプロイを再試行してください。

REMOVE_COMPONENT_ERROR

このエラーは、デプロイ中に nucleus がコンポーネントを削除できない場合に表示されることがあります。ログでエラーの詳細を確認してから、デプロイを再試行してください。

COMPONENT_BOOTSTRAP_TIMEOUT

このエラーは、設定されたタイムアウトよりもコンポーネントのブートストラップタスクが長い時間を要した場合に表示されることがあります。タイムアウトの値を引き上げるか、ブートストラップタスクの実行時間を短くしてから、デプロイを再試行してください。

COMPONENT_BOOTSTRAP_ERROR

このエラーは、コンポーネントのブートストラップタスクにエラーがある場合に表示されることがあります。ログでエラーの詳細を確認してから、デプロイを再試行してください。

COMPONENT_CONFIGURATION_NOT_VALID

このエラーは、nucleus がコンポーネントのデプロイされた設定を検証できない場合に表示されることがあります。ログでエラーの詳細を確認してから、デプロイを再試行してください。

デバイスエラー

IO_WRITE_ERROR

このエラーは、ファイルへの書き込み時に表示されることがあります。詳細については、ログを確認してください。

IO_READ_ERROR

このエラーは、ファイルからの読み取り時に表示されることがあります。詳細については、ログを確認してください。

DISK_SPACE_CRITICAL

このエラーは、デプロイリクエストを完了するのに十分なディスク容量がない場合に表示されることがあります。少なくとも 20 MB の空き容量、またはより大きなアーティファクトを保持するのに十分な容量が必要です。ディスク容量をある程度解放してから、デプロイを再試行してください。

IO_FILE_ATTRIBUTE_ERROR

このエラーは、既存のファイルサイズをファイルシステムから取得できない場合に表示されることがあります。詳細については、ログを確認してください。

SET_PERMISSION_ERROR

このエラーは、ダウンロードしたアーティファクトまたはアーティファクトディレクトリに許可を設定できない場合に表示されることがあります。詳細については、ログを確認してください。

IO_UNZIP_ERROR

このエラーは、アーティファクトを解凍できない場合に表示されることがあります。詳細については、ログを確認してください。

LOCAL_RECIPE_NOT_FOUND

このエラーは、recipe ファイルのローカルコピーが見つからなかった場合に表示されることがあります。デプロイを再試行してください。

LOCAL_RECIPE_CORRUPTED

このエラーは、ダウンロード後に recipe のローカルコピーが変更された場合に表示されることがあります。recipe の既存のコピーを削除して、デプロイを再試行してください。

LOCAL_RECIPE_METADATA_NOT_FOUND

このエラーは、recipe メタデータファイルのローカルコピーが見つからなかった場合に表示されることがあります。デプロイを再試行してください。

LAUNCH_DIRECTORY_CORRUPTED

このエラーは、Greengrass nucleus (`/greengrass/v2/alts/current`) の起動に使用されたディレクトリが、最後に nucleus が起動された後に変更されている場合に表示されることがあります。nucleus を再起動してから、デプロイを再試行してください。

HASHING_ALGORITHM_UNAVAILABLE

このエラーは、デバイスの Java ディストリビューションが必要なハッシュアルゴリズムをサポートしていない場合や、コンポーネント recipe で指定されたハッシュアルゴリズムが有効でない場合に表示されることがあります。

DEVICE_CONFIG_NOT_VALID_FOR_ARTIFACT_DOWNLOAD

このエラーは、デバイス設定にエラーがあり、デプロイが Amazon S3 または Greengrass クラウドからアーティファクトをダウンロードできなかった場合に表示されることがあります。ログに特定の設定エラーがないか確認してから、デプロイを再試行してください。

依存関係のエラー

DOCKER_ERROR

このエラーは、Docker イメージをプルする場合に表示されることがあります。詳細については、その他のエラーコードまたはログを確認してください。

DOCKER_SERVICE_UNAVAILABLE

このエラーは、Greengrass が Docker レジストリにログインできなかった場合に表示されることがあります。特定のエラーがないかをログで確認してから、デプロイを再試行してください。

DOCKER_LOGIN_ERROR

このエラーは、Docker へのログイン時に予期しないエラーが発生した場合に表示されることがあります。特定のエラーがないかをログで確認してから、デプロイを再試行してください。

DOCKER_PULL_ERROR

このエラーは、レジストリから Docker イメージをプルする際に予期しないエラーが発生した場合に表示されることがあります。特定のエラーがないかをログで確認してから、デプロイを再試行してください。

DOCKER_IMAGE_NOT_VALID

このエラーは、リクエストされた Docker イメージが存在しない場合に表示されることがあります。特定のエラーがないかをログで確認し、デプロイを再試行してください。

DOCKER_IMAGE_QUERY_ERROR

使用可能なイメージについて Docker をクエリする際に予期しない障害が発生すると、このエラーが発生する可能性があります。特定のエラーがないかをログで確認し、デプロイを再試行してください。

S3_ERROR

このエラーは、Amazon S3 アーティファクトをダウンロードする場合に表示されることがあります。詳細については、その他のエラーコードまたはログを確認してください。

S3_RESOURCE_NOT_FOUND

このエラーは、Amazon S3 オペレーションが 404 エラーを返した場合に表示されることがあります。詳細については、その他のエラーコードまたはログを確認してください。

S3_BAD_REQUEST

このエラーは、Amazon S3 オペレーションが 400 エラーを返した場合に表示されることがあります。特定のエラーがないかをログで確認し、リクエストを再試行してください。

HTTP エラー

HTTP_REQUEST_ERROR

このエラーは、HTTP リクエストを実行するときにエラーが発生した場合に表示されることがあります。特定のエラーがないかをログで確認してください。

DOWNLOAD_DEPLOYMENT_DOCUMENT_ERROR

このエラーは、デプロイドキュメントのダウンロード中に HTTP エラーが発生した場合に表示されることがあります。特定の HTTP エラーがないかをログで確認してください。

GET_GREENGRASS_ARTIFACT_SIZE_ERROR

このエラーは、パブリックコンポーネントのアーティファクトのサイズを取得するときに HTTP エラーが発生した場合に表示されることがあります。特定の HTTP エラーがないかをログで確認してください。

DOWNLOAD_GREENGRASS_ARTIFACT_ERROR

このエラーは、パブリックコンポーネントのアーティファクトをダウンロードするときに HTTP エラーが発生した場合に表示されることがあります。特定の HTTP エラーがないかをログで確認してください。

ネットワークエラー

NETWORK_ERROR

このエラーは、デプロイ中に接続の問題が発生した場合に表示されることがあります。デバイスのインターネット接続を確認して、デプロイを再試行してください。

nucleus エラー

BAD_REQUEST

このエラーは、AWS クラウドオペレーションが 400 エラーを返した場合に表示されることがあります。どの API がエラーを発生させたかをログで確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

NUCLEUS_VERSION_NOT_FOUND

このエラーは、コアデバイスがアクティブな nucleus のバージョンを見つけられない場合に表示されることがあります。ログでエラーの原因を確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

NUCLEUS_RESTART_FAILURE

このエラーは、nucleus の再起動を必要とするデプロイ中に nucleus が再起動しない場合に表示されることがあります。ローダーログでエラーの原因を確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

INSTALLED_COMPONENT_NOT_FOUND

このエラーは、nucleus がインストールされたコンポーネントを見つけられない場合に表示されることがあります。ログでエラーの原因を確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

DEPLOYMENT_DOCUMENT_NOT_VALID

このエラーは、有効でないデプロイドキュメントをデバイスが受信した場合に表示されることがあります。その他のエラーコードを確認するか、ログでエラーの原因を確認してください。

EMPTY_DEPLOYMENT_REQUEST

このエラーは、デバイスが空のデプロイリクエストを受け取った場合に表示されることがあります。ログでエラーの原因を確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

DEPLOYMENT_DOCUMENT_PARSE_ERROR

このエラーは、デプロイリクエストの形式が、想定される形式と一致しない場合に表示されることがあります。ログでエラーの原因を確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

COMPONENT_METADATA_NOT_VALID_IN_DEPLOYMENT

このエラーは、デプロイリクエストに有効でないコンポーネントメタデータが含まれている場合に表示されることがあります。ログでエラーの原因を確認してから、nucleus ソフトウェアの更

新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

LAUNCH_DIRECTORY_CORRUPTED

Greengrass のデバイスをあるモノグループから別のモノグループに移動し、Greengrass の再起動が必要なデプロイで元のグループに戻したときに、このエラーが表示されることがあります。エラーを解決するには、端末上で Greengrass の起動ディレクトリを再作成してください。

詳細については、「Error: com.aws.greengrass.deployment.exceptions.DeploymentException: Unable to process deployment. Greengrass launch directory is not set up or Greengrass is not set up as a system service」を参照してください。

サーバーエラー

SERVER_ERROR

このエラーは、サービスが現在リクエストを処理できないために、AWS サービスオペレーションが 500 エラーを返した場合に表示されることがあります。後でデプロイを再試行してください。

S3_SERVER_ERROR

このエラーは、Amazon S3 オペレーションが 500 エラーを返した場合に表示されることがあります。詳細については、その他のエラーコードまたはログを確認してください。

クラウドサービスのエラー

RESOLVE_COMPONENT_CANDIDATES_BAD_RESPONSE

このエラーは、Greengrass クラウドサービスが `ResolveComponentCandidates` オペレーションに対して互換性のないレスポンスを送信した場合に表示されることがあります。ログでエラーの原因を確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

DEPLOYMENT_DOCUMENT_SIZE_EXCEEDED

このエラーは、リクエストされたデプロイドキュメントが最大サイズクォータを超えた場合に表示されることがあります。デプロイドキュメントのサイズを小さくして、デプロイを再試行してください。

GREENGRASS_ARTIFACT_SIZE_NOT_FOUND

このエラーは、Greengrass がパブリックコンポーネントアーティファクトのサイズを取得できない場合に表示されることがあります。ログでエラーの原因を確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

DEPLOYMENT_DOCUMENT_NOT_VALID

このエラーは、有効でないデプロイドキュメントをデバイスが受信した場合に表示されることがあります。その他のエラーコードを確認するか、ログでエラーの原因を確認してください。

EMPTY_DEPLOYMENT_REQUEST

このエラーは、デバイスが空のデプロイリクエストを受け取った場合に表示されることがあります。ログでエラーの原因を確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

DEPLOYMENT_DOCUMENT_PARSE_ERROR

このエラーは、デプロイリクエストの形式が、想定される形式と一致しない場合に表示されることがあります。ログでエラーの原因を確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

COMPONENT_METADATA_NOT_VALID_IN_DEPLOYMENT

このエラーは、デプロイリクエストに有効でないコンポーネントメタデータが含まれている場合に表示されることがあります。ログでエラーの原因を確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

一般的なエラー

これらの一般的なエラーには、関連するエラータイプはありません。

DEPLOYMENT_INTERRUPTED

このエラーは、nucleus のシャットダウンや他の外部イベントが原因でデプロイを完了できない場合に表示されることがあります。詳細については、その他のエラーコードまたはログを確認してください。

ARTIFACT_DOWNLOAD_ERROR

このエラーは、アーティファクトのダウンロード中に問題が発生した場合に表示されることがあります。詳細については、その他のエラーコードまたはログを確認してください。

NO_AVAILABLE_COMPONENT_VERSION

このエラーは、コンポーネントバージョンがクラウドもしくはローカルに存在しない場合、または依存関係の解決に競合がある場合に表示されることがあります。詳細については、その他のエラーコードまたはログを確認してください。

COMPONENT_PACKAGE_LOADING_ERROR

このエラーは、ダウンロードしたアーティファクトの処理中にエラーが発生した場合に表示されることがあります。詳細については、その他のエラーコードまたはログを確認してください。

CLOUD_API_ERROR

このエラーは、AWS サービス API の呼び出しでエラーが発生した場合に表示されることがあります。詳細については、その他のエラーコードまたはログを確認してください。

IO_ERROR

このエラーは、デプロイ中に I/O エラーが発生した場合に表示されることがあります。詳細については、その他のエラーコードまたはログを確認してください。

COMPONENT_UPDATE_ERROR

このエラーは、デプロイ中にコンポーネントが更新されない場合に表示されることがあります。その他のエラーコードを確認するか、ログでエラーの原因を確認してください。

未知のエラー

DEPLOYMENT_FAILURE

このエラーは、未チェックの例外がスローされたためにデプロイが失敗した場合に表示されることがあります。ログでエラーの原因を確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

DEPLOYMENT_TYPE_NOT_VALID

このエラーは、デプロイのタイプが有効でない場合に表示されることがあります。ログでエラーの原因を確認してから、nucleus ソフトウェアの更新ページで nucleus の新しいバージョンで問題が修正されているかどうかを確認するか、AWS Support にお問い合わせください。

詳細なコンポーネントのステータスコード

これらのセクションのステータスコードと解決方法を参考にして、Greengrass nucleus バージョン 2.8.0 以降を使用する際のコンポーネントの問題の解決に役立ててください。

このトピックのステータスの多くは、AWS IoT Greengrass Core ログで追加情報を報告します。これらのログは、コアデバイスのローカルファイルシステムに保存されます。個々のコンポーネントごとにログがあります。ログへのアクセスに関する詳細については、「ファイル システム ログをアクセス」を参照してください。

INSTALL_ERROR

これは、インストールスクリプトの実行中にエラーが発生した場合に表示されることがあります。エラーコードはコンポーネントログで報告されます。インストールスクリプトのエラーを確認してコンポーネントを再度デプロイします。

INSTALL_CONFIG_NOT_VALID

このエラーは、recipe の `Install` セクションが有効でないためにコンポーネントのインストールを完了できなかった場合に表示されることがあります。recipe のインストールセクションにエラーがないか確認し、デプロイを再試行します。

INSTALL_IO_ERROR

これは、コンポーネントのインストール中に I/O エラーが発生した場合に表示されることがあります。コンポーネントエラーログでエラーの詳細を確認します。

INSTALL_MISSING_DEFAULT_RUNWITH

このエラーは、AWS IoT Greengrass がコンポーネントのインストール時に使用するユーザーまたはグループを決定できない場合に表示されることがあります。インストール recipe の `runWith` セクションに有効なユーザーまたはグループが含まれていることを確認します。

INSTALL_TIMEOUT

このエラーは、設定したタイムアウト期間内にインストールスクリプトが完了しなかった場合に表示されることがあります。recipe の `Install` セクションで指定された `Timeout` 期間を長くするか、設定されたタイムアウト内に完了するようにインストールスクリプトを変更します。

STARTUP_ERROR

これは、起動スクリプトの実行中にエラーが発生した場合に表示されることがあります。エラーコードはコンポーネントログで報告されます。インストールスクリプトのエラーを確認してコンポーネントを再度デプロイします。

STARTUP_CONFIG_NOT_VALID

このエラーは、recipe の `Startup` セクションが有効でないためにコンポーネントのインストールを完了できなかった場合に表示されることがあります。recipe の起動セクションにエラーがないか確認し、デプロイを再試行します。

STARTUP_IO_ERROR

これは、コンポーネントの起動中に I/O エラーが発生した場合に表示されることがあります。コンポーネントエラーログでエラーの詳細を確認します。

STARTUP_MISSING_DEFAULT_RUNWITH

このエラーは、AWS IoT Greengrass がコンポーネントの実行時に使用するユーザーまたはグループを決定できない場合に表示されることがあります。起動 recipe の `runWith` セクションに有効なユーザーまたはグループが含まれていることを確認します。

STARTUP_TIMEOUT

このエラーは、設定したタイムアウト期間内に起動スクリプトが完了しなかった場合に表示されることがあります。recipe の `Startup` セクションで指定された `Timeout` 期間を長くするか、設定されたタイムアウト内に完了するように起動スクリプトを変更します。

RUN_ERROR

これは、コンポーネントスクリプトの実行中にエラーが発生した場合に表示されることがあります。エラーコードはコンポーネントログで報告されます。実行スクリプトのエラーを確認してコンポーネントを再度デプロイします。

RUN_MISSING_DEFAULT_RUNWITH

このエラーは、AWS IoT Greengrass がコンポーネントの実行時に使用するユーザーまたはグループを決定できない場合に表示されることがあります。実行 recipe の `runWith` セクションに有効なユーザーまたはグループが含まれていることを確認します。

RUN_CONFIG_NOT_VALID

このエラーは、recipe の `Run` セクションが有効でないためにコンポーネントを実行できなかった場合に表示されることがあります。recipe の実行セクションにエラーがないか確認し、デプロイを再試行します。

RUN_IO_ERROR

これは、コンポーネントの実行中に I/O エラーが発生した場合に表示されることがあります。コンポーネントエラーログでエラーの詳細を確認します。

RUN_TIMEOUT

このエラーは、設定したタイムアウト期間内に実行スクリプトが完了しなかった場合に表示されることがあります。recipe の Run セクションで指定された Timeout 期間を長くするか、設定されたタイムアウト内に完了するように実行スクリプトを変更します。

SHUTDOWN_ERROR

これは、コンポーネントスクリプトのシャットダウン中にエラーが発生した場合に表示されることがあります。エラーコードはコンポーネントログで報告されます。シャットダウンスクリプトのエラーを確認してコンポーネントを再度デプロイします。

SHUTDOWN_TIMEOUT

このエラーは、設定したタイムアウト期間内にシャットダウンスクリプトが完了しなかった場合に表示されることがあります。recipe の Shutdown セクションで指定された Timeout 期間を長くするか、設定されたタイムアウト内に完了するように実行スクリプトを変更します。

AWS IoT Greengrass Version 2 リソースのタグ付け

タグを使用すると、AWS IoT Greengrass でリソースを整理および管理できます。タグを使用してリソースにメタデータを割り当てたり、 IAM ポリシーでタグを使用してリソースへの条件付きアクセスを定義したりできます。

> **Note**
>
> 現在のところ、Greengrass リソースタグは AWS IoT 請求グループまたはコスト配分レポートではサポートされていません。

AWS IoT Greengrass V2 でのタグの使用

タグを使用して、目的、所有者、環境、またはユースケースのその他の分類によって AWS IoT Greengrass リソースを分類できます。同じ型のリソースが多い場合に、タグは特定のリソースをより簡単に識別するのに役立ちます。

タグはそれぞれ、1 つのキーとオプションの 1 つの値で設定されており、どちらもお客様側が定義します。例えば、コアデバイスのタグのセットを定義して、デバイスを所有するお客様がタグを追跡できるようにすることができます。リソースの種類ごとのニーズを合わせて一連のタグキーを作成することをお勧めします。一貫したタグキーセットを使用することで、より簡単にリソースを管理できます。

AWS Management Console を使用したタグ付け

AWS Management Console の [Tag Editor] (タグエディタ) は、AWS のすべてのサービスのリソースに対して、タグを作成、管理するための集中的、統一的な方法を提供します。詳細については、[AWS Resource Groups User Guide] (ユーザーガイド) の[Tag Editor] (タグエディタ) を参照してください。

AWS IoT Greengrass V2 API を使用したタグ付け

また、AWS IoT Greengrass V2 API を使用してタグを操作することもできます。タグを作成する前に、タグの制限に注意してください。詳細については、「AWS 全般のリファレンス」の「タグの命名規則と使用規則」を参照してください。

- リソースの作成時にタグを追加するには、リソースの `tags` プロパティでタグを定義します。

- 既存のリソースにタグを追加したり、タグ値を更新するには、TagResource オペレーションを使用します。
- リソースからタグを削除するには、UntagResource オペレーションを使用します。
- リソースに関連付けられたタグを取得するには、ListTagsForResource オペレーションを使用するか、リソースを取得してその `tags` プロパティを調べます。

次の表は、AWS IoT Greengrass V2 API を使ってタグを付けることができるリソースと、それに対応する `Create` および `Describe` または `Get` オペレーションを示しています。

タグ付け可能な AWS IoT Greengrass V2 リソース

リソース	作成オペレーション	オペレーションを説明または取得する
コアデバイス	なし。デバイスで AWS IoT Greengrass Core ソフトウェアを実行して、コアデバイスを作成します。	GetCoreDevice
コンポーネント	CreateComponentVersion	DescribeComponent、GetComponent
デプロイ	CreateDeployment	GetDeployment

ビューで次のオペレーションを使用して、タグ付けをサポートしているリソースのタグを管理します。

- TagResource - リソースにタグを追加するか、既存のタグの値を更新します。
- ListTagsForResource - リソースのタグをリストします。
- UntagResource - リソースからタグを削除します。

リソースに対するタグの追加または削除は随時行うことができます。タグキーの値を変更するには、同じキーと新しい値を定義するリソースにタグを追加します。以前の値は新しい値により上書きされます。値を空の文字列に設定することはできますが、値を null に設定することはできません。

リソースを削除すると、そのリソースに関連付けられているすべてのタグも削除されます。

IAM ポリシーでのタグの使用

IAM ポリシーでは、リソースタグを使用して、ユーザーのアクセスとアクセス許可を制御できます。例えば、ポリシーにより、ユーザーは特定のタグがあるリソースのみを作成できます。ポリシーにより、特定のタグを持つリソースをユーザーが作成または変更できないよう制限することもできます。

> **Note**
>
> タグを使用してリソースへのユーザーのアクセスを許可または拒否する場合は、ユーザーが同じリソースに対してそれらのタグを追加または削除する機能を拒否する必要があります。そうしないと、ユーザーはそのリソースのタグを変更することで、制限を回避してリソースにアクセスできてしまいます。

ポリシーステートメントの Condition 要素 (Condition ブロックとも呼ばれる) の次の条件コンテキストキーと値を使用できます。

`greengrassv2:ResourceTag/`*`tag-key`*`: `*`tag-value`*

特定のタグを持つリソースに対してアクションを許可または拒否します。

`aws:RequestTag/`*`tag-key`*`: `*`tag-value`*

タグ付け可能なリソースを作成または変更するときに、特定のタグを使用する (または使用しない) よう要求します。

`aws:TagKeys: [`*`tag-key`*`, ...]`

タグ付け可能なリソースを作成または変更するときに、特定のタグキーセットを使用する (または使用しない) よう要求します。

> **Note**
>
> IAM ポリシーの条件コンテキストキーと値は、必須パラメータとしてタグ付け可能なリソースを持つアクションにのみ適用されます。例えば、ListCoreDevices に対してタグベースの条件付きアクセスを設定できます。

詳細については、[IAM User Guide] (IAM ユーザーガイド) の[Controlling access to AWS resources using resource tags] (リソースタグを使用したリソースへのアクセスの制御) および [IAM JSON policy reference] (IAM JSON ポリシーリファレンス) を参照してください。

AWS CloudFormation での AWS IoT Greengrass リソースの作成

AWS IoT Greengrass は、リソースとインフラストラクチャの作成と管理の所要時間を短縮できるように AWS リソースをモデル化して設定するためのサービスである AWS CloudFormation と統合されています。必要なすべての AWS リソース (コンポーネントバージョンやデプロイなど) を説明するテンプレートを作成すれば、AWS CloudFormation がお客様に代わってこれらのリソースのプロビジョニングや設定を処理します。

AWS CloudFormation を使用すると、テンプレートを再利用して AWS IoT Greengrass リソースを同じように繰り返してセットアップできます。リソースを一度記述するだけで、同じリソースを複数の AWS アカウント とリージョンで何度でもプロビジョニングできます。

AWS IoT Greengrass テンプレートと AWS CloudFormation テンプレート

AWS IoT Greengrass および関連サービスのリソースをプロビジョニングして設定するには、AWS CloudFormation テンプレートについて理解しておく必要があります。テンプレートは、JSON またはYAMLでフォーマットされたテキストファイルです。これらのテンプレートには、AWS CloudFormation スタックにプロビジョニングしたいリソースを記述します。JSONやYAMLに不慣れな方は、AWS CloudFormation Designerを使えば、 AWS CloudFormation テンプレートを使いこなすことができます。詳細については、「AWS CloudFormation ユーザーガイド」の「AWS CloudFormation Designer とは」を参照してください。

AWS IoT Greengrass は AWS CloudFormation でのコンポーネントバージョンとデプロイの作成をサポートします。コンポーネントバージョンやデプロイの JSON テンプレートと YAML テンプレートの例を含む詳細については、「AWS CloudFormation ユーザーガイド」の「AWS IoT Greengrass リソースタイプのリファレンス」を参照してください。

ComponentVersion のテンプレートの例

シンプルなコンポーネントバージョン用の YAML テンプレートを次に示します。JSON recipe には、読みやすいように改行が追加されています。

```
Parameters:
  ComponentVersion:
```

```
    Type: String
Resources:
  TestSimpleComponentVersion:
    Type: AWS::GreengrassV2::ComponentVersion
    Properties:
      InlineRecipe: !Sub
        - "{\n
            \"RecipeFormatVersion\": \"2020-01-25\",\n
            \"ComponentName\": \"component1\",\n
            \"ComponentVersion\": \"${ComponentVersion}\",\n
            \"ComponentType\": \"aws.greengrass.generic\",\n
            \"ComponentDescription\": \"This\",\n
            \"ComponentPublisher\": \"You\",\n
            \"Manifests\": [\n
            {\n
              \"Platform\": {\n
              \"os\": \"darwin\"\n
            },\n
            \"Lifecycle\": {},\n
            \"Artifacts\": []\n
           },\n
           {\n
             \"Lifecycle\": {},\n
             \"Artifacts\": []\n
           }\n
          ],\n
          \"Lifecycle\": {\n
          \"install\": {\n
            \"script\": \"yuminstallpython\"\n
          }\n
         }\n
        }"
        - { ComponentVersion: !Ref ComponentVersion }
```

デプロイテンプレートの例

デプロイ用のシンプルなテンプレートを定義する YAML ファイルを次に示します。

```
Parameters:
  ComponentVersion:
    Type: String
  TargetArn:
    Type: String
```

```
Resources:
  TestDeployment:
    Type: AWS::GreengrassV2::Deployment
    Properties:
      Components:
        component1:
          ComponentVersion: !Ref ComponentVersion
      TargetArn: !Ref TargetArn
      DeploymentName: CloudFormationIntegrationTest
      DeploymentPolicies:
        FailureHandlingPolicy: DO_NOTHING
        ComponentUpdatePolicy:
          TimeoutInSeconds: 5000
          Action: SKIP_NOTIFY_COMPONENTS
        ConfigurationValidationPolicy:
          TimeoutInSeconds: 30000
Outputs:
  TestDeploymentArn:
    Value: !Sub
      - arn:${AWS::Partition}:greengrass:${AWS::Region}:${AWS::AccountId}:deployments:
${DeploymentId}
      - DeploymentId: !GetAtt TestDeployment.DeploymentId
```

AWS CloudFormation の詳細はこちら

AWS CloudFormation の詳細については、以下のリソースを参照してください。

- AWS CloudFormation
- AWS CloudFormation ユーザーガイド
- AWS CloudFormation API リファレンス
- AWS CloudFormation コマンドラインインターフェイスユーザーガイド

オープンソース AWS IoT Greengrass Core ソフトウェア

AWS IoT Greengrass Core ソフトウェアの AWS IoT Greengrass Version 2 エッジランタイム (nucleus) と他のコンポーネントはオープンソースです。つまり、コードを確認して、アプリケーションのインタラクションをトラブルシューティングできます。お客様の特定のソフトウェアとハードウェアニーズを満たすため、AWS IoT Greengrass Core ソフトウェアをカスタマイズして拡張することもできます。

AWS IoT Greengrass Core ソフトウェアのオープンソースリポジトリの詳細については、「」の「aws-greengrass organization」を参照してください GitHub。オープンソースソフトウェアの使用は、対応する GitHub リポジトリ のオープンソースライセンスによって管理されます。

オープンソースライセンスの対象にならない AWS IoT Greengrass Core ソフトウェアとコンポーネントの使用は、AWS Greengrass Core Software License によって管理されます。

AWS IoT Greengrass V2 デベロッパーガイドのドキュメント履歴

次の表は、 の今回のリリースの内容をまとめたものです AWS IoT Greengrass Version 2。

- API バージョン: 2020 年 11 月 30 日

変更	説明	日付
AWS IoT Device Tester v4.9.2 と GGV2Q v2.5.2 のリリース	IDT for AWS IoT Greengrass V2 のバージョン 4.9.2 が利用可能になりました。このリリースには AWS IoT Greengrass V2 認定スイート (GGV2Q) v2.5.2 が含まれており、Greengrass nucleus バージョン 2.12.0、2.11.0、2.10.0、2.9.5 をサポートしています。	2024 年 3 月 18 日
Lookout for Vision エッジエージェント v1.2.0 がリリースされました	Lookout for Vision エッジエージェント v1.2.0 が利用可能になりました。	2024 年 3 月 11 日
AWS IoT Greengrass Core v2.12.2 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.12.2 が提供され、AWSが提供するコンポーネントが更新されます。	2024 年 2 月 15 日
シャドウマネージャー v2.3.6 がリリースされました	シャドウマネージャー v2.3.6 が利用可能になりました。このリリースでは、デバイスのオフライン中に AWS クラウド 更新によって削除されたシャドウプロパティが、接続	2024 年 2 月 14 日

	の回復後もローカルシャドウに引き続き存在する問題が修正されています。	
Lambda ランチャー v2.0.13 がリリースされました	Lambda ランチャーコンポーネントのバージョン 2.0.13 を利用できます。このリリースには、一般的なバグ修正と改善が含まれています。	2024 年 2 月 14 日
ディスクスプーラ v1.0.3 のリリース	ディスクスプーラコンポーネント v1.0.3 が利用可能になりました。このリリースでは、データベース接続を再利用することでパフォーマンスが向上します。	2024 年 2 月 14 日
Lookout for Vision エッジエージェント v1.1.9 をリリース	Lookout for Vision エッジエージェント v1.1.9 が利用可能になりました。	2024 年 1 月 17 日
Greengrass Development Kit CLI v1.6.2	Greengrass Development Kit CLI のバージョン 1.6.2 が利用可能になりました。このバージョンでは、相対パスが原因で Windows gradlew.bat が動作しない問題が修正されています。このバージョンには、追加の改善も含まれています。	2024 年 1 月 16 日

新しい CloudTrail データイベント	AWS CloudTrail データイベントをログに記録して、コンポーネントの取得やデプロイの設定などのリソースオペレーションに関する情報を取得できるようになりました。これらのイベントを使用して、Greengrass デバイスのオペレーションに関するインサイトを取得します。	2023 年 12 月 20 日
Lookout for Vision エッジエージェント v1.1.8 がリリースされました	Lookout for Vision エッジエージェント v1.1.8 が利用可能になりました。	2023 年 12 月 12 日
ストリームマネージャー v2.1.12 がリリースされました	ストリームマネージャー v2.1.12 が利用可能になりました。このリリースでは、Greengrass が AWS サービス呼び出し用の認証情報のセットを選択するために使用する順序が変更されています。	2023 年 12 月 8 日
MQTT ブリッジ v2.3.1 がリリースされました	MQTT ブリッジ v2.3.1 が利用可能になりました。このリリースでは、ローカル MQTT クライアントが切断ループに入るというまれな問題が修正されています。	2023 年 12 月 8 日

ディスクスプーラ v1.0.2 のリリース	ディスクスプーラコンポーネント v1.0.2 が利用可能になりました。このリリースでは、特定のケースで MQTT メッセージ形式フィールドが保持されない問題が修正されています。	2023 年 12 月 8 日
クライアントデバイス認証コンポーネント v2.4.5 がリリースされました	クライアントデバイス認証コンポーネント v2.4.5 が利用可能になりました。このリリースでは、選択ルールでモノの名前の末尾にワイルドカードのサポートが追加され、場合によっては証明書が新しい接続情報で更新されない問題が修正されています。	2023 年 12 月 8 日
AWS IoT Greengrass Core v2.12.1 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.12.1 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 12 月 8 日
Greengrass Development Kit CLI v1.6.1	Greengrass Development Kit CLI のバージョン 1.6.1 が利用可能になりました。このバージョンには、バグ修正と機能向上が含まれています。	2023 年 12 月 6 日
レシピの検証	コンポーネントバージョンの作成時にコンポーネントレシピを検証するレシピ検証機能を追加しました。	2023 年 11 月 16 日

パブリッシャーがサポートするコンポーネント	AWS IoT Greengrass は、パブリッシャーがサポートするコンポーネントを提供するようになりました。これらのコンポーネントは、サードパーティーベンダーによって開発、提供、および提供されます。	2023 年 11 月 16 日
Greengrass Testing Framework v1.2.0 をリリース	Greengrass Testing Framework v1.2.0 が利用可能になりました。	2023 年 11 月 15 日
Greengrass Development Kit CLI v1.6.0	Greengrass Development Kit CLI のバージョン 1.6.0 が利用可能になりました。このバージョンでは、 `component build`および `component publish` コマンド中に Greengrass レシピスキーマに対するレシピ検証チェックが追加されます。この更新により、デベロッパーはコンポーネント作成プロセスの早い段階でコンポーネントレシピ内の実用的な問題を特定できます。このバージョンでは、`test-e2e init` コマンドでプルダウンできる信頼度テストスイートがテンプレートに追加されます。この信頼度テストスイートには、基本コンポーネントテストのニーズに合わせて使用および拡張できる 8 つの汎用テストが含まれています。	2023 年 11 月 15 日

AWS IoT Device Tester v4.9.1 が Greengrass nucleus バージョン 2.12.0 をサポート	IDT for AWS IoT Greengrass V2 のバージョン 4.9.1 では、Greengrass nucleus バージョン 2.12.0 がサポートされるようになりました。	2023 年 11 月 7 日
AWS IoT Greengrass Core v2.12.0 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.12.0 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 11 月 7 日
VPC で Greengrass コアデバイスを運用する	VPC で Greengrass コアデバイスを操作できます。この機能により、パブリックインターネットアクセスなしで VPC にデプロイを実行できます。	2023 年 11 月 3 日
Greengrass CLI v2.12.0 をリリース	Greengrass CLI コンポーネント v2.12.0 が利用可能になりました。	2023 年 10 月 30 日
ストリームマネージャー v2.1.10 がリリースされました	ストリームマネージャー v2.1.10 が利用可能になりました。このリリースでは、HTTPS プロキシ設定が Greengrass CA 証明書チェーンを信頼しない問題が修正されています。	2023 年 10 月 26 日

Lambda ランチャー v2.0.12 がリリースされました	Lambda ランチャーコンポーネントのバージョン 2.0.12 を利用できます。このリリースでは、前のプロセスが正しく停止しなかった場合に Lambda ランチャーがエラーをスローすることがある問題が修正されています。	2023 年 10 月 26 日
Greengrass Development Kit CLI v1.5.0	Greengrass Development Kit CLI のバージョン 1.5.0 が利用可能になりました。このバージョンでは、 `build_system` が の場合に`excludes`ビルドオプションで認識されるパターンが更新されます`zip`。このバージョンでは、ワイルドカード文字に基づいてパス名と一致する glob パターンが認識されるようになりました。これにより、除外するディレクトリのカスタム仕様が有効になります。	2023 年 10 月 26 日
Lookout for Vision エッジエージェント v1.1.7 がリリースされました	Lookout for Vision エッジエージェント v1.1.7 が利用可能になりました。	2023 年 10 月 24 日
シャドウマネージャー v2.3.4 がリリースされました	シャドウマネージャー v2.3.4 が利用可能になりました。このリリースでは、null および空のシャドウ状態ドキュメントのサポートが追加されました。	2023 年 10 月 18 日

ログマネージャー v2.3.6 がリリースされました	ログマネージャーコンポーネント v2.3.6 が利用可能になりました。	2023 年 10 月 18 日
ローカルデバッグコンソール v2.4.0 をリリース	ローカルデバッグコンソールコンポーネント v2.4.0 が利用可能になりました。	2023 年 10 月 18 日
Lambda マネージャー v2.3.1 がリリースされました	Lambda マネージャーコンポーネント v2.3.1 が利用可能になりました。	2023 年 10 月 18 日
Greengrass CLI v2.11.3 をリリース	Greengrass CLI コンポーネント v2.11.3 が利用可能になりました。	2023 年 10 月 18 日
AWS IoT Greengrass Core v2.11.3 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.11.3 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 10 月 18 日
セキュアトンネリング v1.0.17 がリリースされました	セキュアトンネリング v1.0.17 が利用可能になりました。	2023 年 10 月 4 日

Greengrass Development Kit CLI v1.4.0	Greengrass Development Kit CLI のバージョン 1.4.0 を利用できます。このバージョンでは、既存の GDK 設定ファイル内のフィールドを変更するインタラクティブプロンプトを起動する新しい `config` コマンドが追加されています。このバージョンでは、レシピサイズが Greengrass の要件 (16000 バイト未満) 以内であることを確認してから続行するように `gdk component build` および `gdk component publish` コマンドが変更されています。	2023 年 10 月 2 日
Moquette MQTT 3.1.1 ブローカー v2.3.5 がリリースされました	Moquette MQTT 3.1.1 ブローカーコンポーネントのバージョン 2.3.5 が利用可能になりました。このバージョンでは Moquette をバージョン 0.17 に更新しています。	2023 年 9 月 28 日
MQTT ブリッジ v2.3.0 がリリースされました	MQTT ブリッジ v2.3.0 が利用可能になりました。このリリースでは、 AWS IoT Core とローカル MQTT ソース間のブリッジ用の MQTT 5 サポートを追加します。	2023 年 9 月 28 日
Lookout for Vision Edge Agent v1.1.6 をリリース	Lookout for Vision Edge Agent v1.1.6 が利用可能になりました。	2023 年 9 月 27 日

Lambda manager v2.3.0 をリリース	Lambda manager コンポーネント v2.3.0 を利用できます。	2023 年 9 月 15 日
Lambda ランチャー v2.0.11 がリリースされました	Lambda ランチャーコンポーネントのバージョン 2.0.11 を利用できます。このバージョンでは、Lambda Manager 2.3.0 がサポートされています。	2023 年 9 月 15 日
Moquette MQTT 3.1.1 ブローカー v2.3.4 がリリースされました	Moquette MQTT 3.1.1 ブローカーコンポーネントのバージョン 2.3.4 が利用可能になりました。	2023 年 9 月 1 日
Greengrass テストフレームワーク	GTF は、 end-to-end オートメーションをサポートするための構成要素のコレクションです。これにより、 AWS IoT Greengrass Version 2 をご利用の内部のお客様は、サービスチームがソフトウェアの変更、自動承認、および品質保証の目的を満たすために使用するものと同じテストフレームワークを使用できます。	2023 年 8 月 11 日
AWS IoT Greengrass Core v2.11.2 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.11.2 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 8 月 9 日

Greengrass Development Kit CLI v1.3.0	Greengrass Development Kit CLI のバージョン 1.3.0 を利用できます。このバージョンでは、Open Test Framework を使用したコンポーネントの end-to-end テストをサポートする新しい `test-e2e` コマンドが追加されています。	2023 年 7 月 21 日
AWS IoT Greengrass Core v2.11.1 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.11.1 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 7 月 21 日
ディスクスプーラ v1.0.0 がリリースされました	ディスクスプーラコンポーネント v1.0.0 が利用可能です。	2023 年 6 月 28 日
AWS IoT Greengrass Core v2.11.0 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.11.0 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 6 月 28 日
AWS IoT Greengrass Core v2.10.3 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.10.3 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 6 月 21 日
AWS IoT Greengrass Core v2.10.2 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.10.2 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 6 月 5 日

AWS IoT Greengrass Core v2.10.1 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.10.1 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 5 月 11 日
AWS IoT Greengrass Core v2.10.0 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.10.0 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 5 月 9 日
SageMaker Edge Manager の廃止	Amazon SageMaker Edge Manager コンポーネントは 2024 年 4 月 26 日に廃止されます。	2023 年 4 月 28 日
AWS IoT Greengrass Core v2.9.6 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.6 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 4 月 20 日
ログマネージャー v2.3.2 をリリースしました	ログマネージャーコンポーネント v2.3.2 が利用可能になりました。	2023 年 4 月 19 日

ストリームマネージャー v2.1.4 をリリースしました	ストリームマネージャー v2.1.4 が利用可能になりました。このリリースでは、1 つのバッチ内で同じタイムスタンプを持つ同じプロパティアセットのエントリが SiteWise API `ConflictingOperationException` から返され、ストリームマネージャーが継続的に再試行する問題が修正されています。また、このリリースでは、デフォルトの接続タイムアウトを 3 秒から 1 分に更新しました。	2023 年 4 月 13 日
Greengrass Development Kit CLI v1.2.3	Greengrass Development Kit CLI のバージョン 1.2.3 が利用可能になりました。このバージョンには、バグ修正が含まれています。	2023 年 4 月 13 日
クライアントデバイス認証コンポーネント v2.4.0 をリリースしました	クライアントデバイス認証コンポーネントの v2.4.0 が利用可能になりました。このリリースでは、Greengrass クライアントデバイスダッシュボードに表示できる操作メトリックを生成するクライアントデバイス認証のサポートが追加されています。	2023 年 4 月 10 日
Greengrass Development Kit CLI v1.2.2	Greengrass Development Kit CLI のバージョン 1.2.2 が利用可能になりました。このバージョンには、機能向上とバグ修正が含まれています。	2023 年 4 月 7 日

AWS IoT Greengrass Core v2.9.5 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.5 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 3 月 30 日
ストリームマネージャー v2.1.3 をリリース	ストリームマネージャー v2.1.3 が利用可能になりました。このリリースでは、Windows OS を SYSTEM ユーザーとして実行する際に発生する、起動上の問題が修正されています。	2023 年 3 月 7 日
Modbus-RTU プロトコルアダプタ v2.1.5 をリリース	Modbus-RTU プロトコルアダプタコンポーネント v2.1.5 を利用できます。このリリースでは、`ReadDiscreteInput` オペレーションに関する問題を修正しました。	2023 年 3 月 7 日
クライアントデバイス認証コンポーネント v2.3.2 をリリース	クライアントデバイス認証コンポーネントの v2.3.2 が利用可能です。このリリースでは、ホスト名情報のキャッシュが新たにサポートされ、コンポーネントがオフラインで再起動する際に、証明書のサブジェクトが正しく生成されるようになりました。	2023 年 3 月 7 日
AWS IoT Device Tester v4.7.0 が Greengrass nucleus バージョン 2.9.4 をサポート	IDT for AWS IoT Greengrass V2 のバージョン 4.7.0 が Greengrass nucleus バージョン 2.9.4 をサポートするようになりました。	2023 年 3 月 2 日

Greengrass コマンドラインインターフェイス v1.2.0 をリリース	Greengrass コマンドラインインターフェイス v1.2.0 が利用可能です。	2023 年 2 月 28 日
AWS IoT Greengrass Core v2.9.4 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.4 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 2 月 24 日
シャドウマネージャー v2.3.1 をリリース	シャドウマネージャー v2.3.1 が利用可能になりました。このリリースでは、クラウドシャドウアップデートの同期を妨げる可能性のある状態が修正されています。このリリースでは、名前付きシャドウ同期設定の変更が 1 つの名前付きシャドウにのみ適用される問題も修正されています。	2023 年 2 月 21 日
AWS IoT Device Tester v4.7.0 が Greengrass nucleus バージョン 2.9.3 をサポート	IDT for AWS IoT Greengrass V2 のバージョン 4.7.0 が Greengrass nucleus バージョン 2.9.3 をサポートするようになりました。	2023 年 2 月 9 日
IAM ベストプラクティスの更新	IAM のベストプラクティスに合わせてガイドを更新しました。詳細については、「IAM のセキュリティのベストプラクティス」を参照してください。	2023 年 2 月 3 日

AWS IoT Greengrass Core v2.9.3 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.3 が提供され、AWSが提供するコンポーネントが更新されます。	2023 年 2 月 1 日
ログマネージャー v2.3.1 をリリース	ログマネージャー v2.3.1 を利用できます。	2023 年 1 月 27 日
AWS IoT Device Tester v4.7.0 が Greengrass nucleus バージョン 2.9.2 をサポート	IDT for AWS IoT Greengrass V2 のバージョン 4.7.0 が Greengrass nucleus バージョン 2.9.2 をサポートするようになりました。	2023 年 1 月 3 日
シャドウマネージャー v2.3.0 をリリース	シャドウマネージャー v2.3.0 が利用可能になりました。このリリースでは、デバイスによって Greengrass デバイスのプライベートキーがハードウェアセキュリティモジュールに保存されている場合にシャドウが同期されない可能性がある問題が修正されています。	2022 年 12 月 29 日
AWS IoT フリートプロビジョニングプラグイン v1.2.0 をリリース	AWS IoT フリートプロビジョニングプラグイン v1.2.0 が利用可能になりました。今回のリリースでは、プライベートキーの設定可能なパスを使用した証明書署名リクエストを介するデバイスプロビジョニングのサポートが追加されています。	2022 年 12 月 22 日

AWS IoT Greengrass Core v2.9.2 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.2 が提供され、AWSが提供するコンポーネントが更新されます。	2022 年 12 月 22 日
AWS IoT Device Tester v4.7.0 と GGV2Q v2.5.0 のリリース	IDT for AWS IoT Greengrass V2 のバージョン 4.7.0 が利用可能になりました。このリリースには AWS IoT Greengrass V2 認定スイート (GGV2Q) v2.5.0 が含まれており、Greengrass nucleus バージョン 2.9.1、2.9.0、2.8.1、2.8.0、2.7.0、および 2.6.0 をサポートしています。	2022 年 12 月 13 日
シャドウマネージャー v2.2.4 をリリース	ローカルのシャドウドキュメントを更新するときに、シャドウのサイズの検証がクラウドと一貫しない問題を修正しました。デプロイが設定ノードに対して RESET を実行する場合に、シャドウマネージャーが設定の更新のリッスンを停止する問題も修正しました。	2022 年 12 月 8 日
Lookout for Vision Edge Agent 1.1.1 をリリース	Lookout for Vision Edge Agent コンポーネント v1.1.1 が利用可能になりました。	2022 年 12 月 5 日
ログマネージャー v2.3.0 をリリース	ログマネージャーコンポーネント v2.3.0 が利用可能になりました。	2022 年 11 月 18 日

AWS IoT Device Tester v4.5.11 が Greengrass nucleus バージョン 2.9.1 をサポート	IDT for AWS IoT Greengrass V2 のバージョン 4.5.11 では、Greengrass nucleus バージョン 2.9.1 がサポートされるようになりました。	2022 年 11 月 18 日
AWS IoT Greengrass Core v2.9.1 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.1 が提供され、AWSが提供するコンポーネントが更新されます。	2022 年 11 月 18 日
AWS IoT Device Tester v4.5.11 が Greengrass nucleus バージョン 2.9.0 をサポート	IDT for AWS IoT Greengrass V2 のバージョン 4.5.11 では、Greengrass nucleus バージョン 2.9.0 がサポートされるようになりました。	2022 年 11 月 17 日
ストリームマネージャー v2.1.2 をリリース	ストリームマネージャー v2.1.2 が利用可能になりました。このリリースでは、英語以外の言語を使用する Windows OS における問題が修正されています。	2022 年 11 月 15 日
AWS IoT Greengrass Core v2.9.0 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.9.0 が提供され、AWSが提供するコンポーネントが更新されます。	2022 年 11 月 15 日
AWS IoT Device Tester v4.5.11 が Greengrass nucleus バージョン 2.8.1 をサポート	IDT for AWS IoT Greengrass V2 のバージョン 4.5.11 では、Greengrass nucleus バージョン 2.8.1 がサポートされるようになりました。	2022 年 10 月 19 日

AWS IoT Device Tester v4.5.11 と GGV2Q v2.4.1 のリリース	IDT for AWS IoT Greengrass V2 のバージョン 4.5.11 が利用可能になりました。このリリースには AWS IoT Greengrass V2 認定スイート (GGV2Q) v2.4.1 が含まれており、Greengrass nucleus バージョン 2.8.0、2.7.0、および 2.6.0 をサポートしています。	2022 年 10 月 13 日
AWS IoT Greengrass Core v2.8.1 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.8.1 が提供され、AWSが提供するコンポーネントが更新されます。	2022 年 10 月 13 日
AWS IoT Greengrass Core v2.8.0 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.8.0 が提供され、AWSが提供するコンポーネントが更新されます。	2022 年 10 月 7 日
デプロイ AWS CloudFormation のサポートを追加	AWS CloudFormation がリソースとしての AWS IoT Greengrass デプロイをサポートするようになりました。	2022 年 10 月 6 日

SageMaker Edge Manager v1.3.0 のリリース	Amazon SageMaker Edge Manager コンポーネント v1.3.0 が利用可能になりました。このリリースでは、TensorRT モデルキャッシュのディスクサイズを設定するこのコンポーネントのサポートが追加され、予測の同時実行を改善して、GPU などのデバイスアクセラレータエンジンをより有効に活用できるようになりました。	2022 年 9 月 1 日
プロセス間通信 (IPC) クライアント V2 を使用する	IPC クライアント V2 についての情報が追加されました。これにより、IPC オペレーションを使用するためのコードの記述量が削減し、IPC クライアント V1 で発生する可能性のある、一般的なエラーを回避することができます。	2022 年 8 月 12 日
AWS IoT Device Tester v4.5.8 と GGV2Q v2.4.0 のリリース	IDT for AWS IoT Greengrass V2 のバージョン 4.5.8 が利用可能になりました。このリリースには AWS IoT Greengrass V2 認定スイート (GGV2Q) v2.4.0 が含まれており、Greengrass nucleus バージョン 2.7.0、2.6.0、および 2.5.6 をサポートしています。	2022 年 8 月 12 日

SageMaker Edge Manager v1.2.0 のリリース	Amazon SageMaker Edge Manager コンポーネント v1.2.0 が利用可能になりました。このリリースでは、Amazon S3 にアップロードする SageMaker Neo コンパイル済みモデルを自動的に取得するこのコンポーネントのサポートが追加されています。そのため、デプロイを作成することなく新しいモデルを AWS IoT Greengrass デプロイできます。	2022 年 8 月 3 日
AWS IoT Device Tester v4.5.3 が Greengrass nucleus バージョン 2.7.0 をサポート	IDT for AWS IoT Greengrass V2 のバージョン 4.5.3 では、Greengrass nucleus バージョン 2.7.0 がサポートされるようになりました。	2022 年 8 月 1 日
ストリームマネージャー v2.1.0 をリリース	ストリームマネージャー v2.1.0 が利用可能になりました。このリリースには、テレメトリメトリクスを Amazon に送信するためのサポートが含まれています EventBridge。	2022 年 7 月 28 日
AWS IoT Greengrass Core v2.7.0 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.7.0 が提供され、AWSが提供するコンポーネントが更新されます。これには、テレメトリメトリクスを Amazon に送信するためのサポートが含まれます EventBridge。	2022 年 7 月 28 日

IoT SiteWise パブリッシャー v2.2.0 をリリース	IoT SiteWise パブリッシャーコンポーネント v2.2.0 が利用可能になりました。このリリースでは、コンポーネントを更新してデータを圧縮してから AWS IoT SiteWise サービスに送信します。これにより、帯域幅の使用量が最大 75% 削減されます。	2022 年 7 月 19 日
チュートリアル: クライアントデバイスを操作するコンポーネントを開発するには	「チュートリアル: MQTT 経由でローカル IoT デバイスを操作する」に、新しいモジュールを追加しました。これに従うことで、クライアントデバイスシャドウを操作するコンポーネントの開発方法を確認できます。	2022 年 7 月 18 日
ローカルの MQTT ブローカーを選択する	クライアントデバイスをコアデバイスに接続するローカル MQTT ブローカーの、選択方法に関する情報を追加しました。	2022 年 7 月 18 日
AWS IoT Device Tester v4.5.3 が Greengrass nucleus バージョン 2.6.0 をサポート	IDT for AWS IoT Greengrass V2 のバージョン 4.5.3 が Greengrass nucleus バージョン 2.6.0 をサポートするようになりました。	2022 年 1 月 29 日

AWS IoT Greengrass Core v2.6.0 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.6.0 が提供され、AWSが提供するコンポーネントが更新されます。これには、クライアントデバイスシャドウと、クライアントデバイス用のローカル MQTT 5 ブローカーのサポートが含まれます。また、ローカルでのトピックの公開/サブスクライブ用のワイルドカード、コンポーネント設定での recipe 変数、および IPC 認証ポリシーでのワイルドカードのサポートも含まれています。これらの機能により、コアデバイスのフリートにデプロイするコンポーネントを、より簡単に開発し設定できるようになります。このリリースには、ローカルでのデプロイとコアデバイス上のコンポーネントを管理する IPC オペレーションを使用するための、コンポーネントに関するサポートも含まれています。	2022 年 6 月 27 日

クライアントデバイスのコンポーネントの更新	クライアントデバイス認証 v2.1.0、MQTT ブローカー (Moquette) v2.1.0、MQTT ブリッジ v2.1.1、およびIP ディテクター v2.1.2を利用可能です。このリリースでは、証明書のローテーションが改善し、MQTT ブローカーのパフォーマンスが向上されており、これらのコンポーネントが設定リセットの更新を処理する方法から生じる問題が修正されています。	2022 年 6 月 14 日
AWS IoT Device Tester v4.5.3 が Greengrass nucleus バージョン 2.5.6 をサポート	IDT for AWS IoT Greengrass V2 のバージョン 4.5.3 が Greengrass nucleus バージョン 2.5.6 をサポートするようになりました。	2022 年 6 月 1 日
AWS IoT Greengrass Core v2.5.6 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.5.6 が提供され、AWSが提供するコンポーネントが更新されます。ECC キーを採用したハードウェアセキュリティモジュールへのサポートが含まれています。その他のバグ修正や改良も含まれます。	2022 年 5 月 31 日

AWS IoT フリートプロビジョニングプラグイン v1.1.0 をリリース	AWS IoT フリートプロビジョニングプラグイン v1.1.0 が利用可能になりました。このリリースでは、Windows デバイスでプラグインを設定するときの、追加のファイルパス形式へのサポートが追加されています。	2022 年 5 月 12 日
新しい Lambda ランタイムをリリース	新しい Lambda ランタイム (Python 3.9、Java 11、NodeJS 14) へのサポートが追加されました。	2022 年 5 月 10 日
コンポーネントの更新を延期する Greengrass コンポーネントを開発	コンポーネント更新のデプロイを延期する Greengrass コンポーネントを開発する方法についてのチュートリアルが追加されました。たとえば、デバイスのバッテリ残量が少ない場合や、中断できないプロセスを実行している最中の場合などに、更新を遅らせることができます。	2022 年 5 月 4 日

CloudWatch メトリクス v3.1.0 および AWS IoT Device Defender v3.1.0 がリリースされました	CloudWatch メトリクスコンポーネント v3.1.0 と AWS IoT Device Defender コンポーネント v3.1.0 を利用できます。これらのリリースでは、HTTPS ネットワークプロキシ設定へのサポートが追加されています。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」および「コアデバイスでの HTTPS プロキシへの信頼を有効にする」を参照してください。	2022 年 4 月 27 日
からの移行 AWS IoT Greengrass Version 1	から への移行に関するガイドを追加しました AWS IoT Greengrass V1 AWS IoT Greengrass V2。	2022 年 4 月 26 日

AWS IoT Device Tester v4.5.3 with GGV2Q v2.3.1 が更新され、IDT v4.5.1 with GGV2Q v2.3.0 がサポートされているバージョンに追加されました	IDT for AWS IoT Greengrass V2 with V2 認定スイート (GGV2Q) AWS IoT Greengrass V2v2.3.1 のバージョン 4.5.3 が更新され、Greengrass nucleus バージョン 2.5.5、2.5.4、2.5.3 のサポートが追加されました。この更新には、サポートされているバージョンとして AWS IoT Greengrass V2 認定スイート (GGV2Q) v2.3.0 を使用する IDT 4.5.1 も含まれています。IDT 4.5.1 with AWS IoT Greengrass V2 認定スイート (GGV2Q) v2.3.0 は Greengrass nucleus バージョン 2.5.3 をサポートしています。	2022 年 4 月 25 日
Modbus-RTU プロトコルアダプタ v2.1.0 をリリース	Modbus-RTU プロトコルアダプタコンポーネント v2.1.0 を利用できます。このリリースには、Modbus RTU デバイスとのシリアル通信を設定するように指定できる新しいパラメータが追加されています。	2022 年 4 月 20 日

CloudWatch メトリクス v2.1.0、Firehose v2.1.0、および Amazon SNS v2.1.0 をリリース	CloudWatch メトリクスコンポーネント v2.1.0、Firehose コンポーネント v2.1.0、および Amazon SNS コンポーネント v2.1.0 を利用できます。これらのリリースでは、HTTPS ネットワークプロキシ設定へのサポートが追加されています。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」および「コアデバイスでの HTTPS プロキシへの信頼を有効にする」を参照してください。	2022 年 4 月 19 日
AWS IoT Device Tester v4.5.3 と GGV2Q v2.3.1 のリリース	IDT for AWS IoT Greengrass V2 のバージョン 4.5.3 が利用可能になりました。このリリースには AWS IoT Greengrass V2 認定スイート (GGV2Q) v2.3.1 が含まれており、Greengrass nucleus バージョン 2.5.5 をサポートしています。	2022 年 4 月 15 日

AWS IoT Greengrass Core v2.5.5 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.5.5 が提供され、AWSが提供するコンポーネントが更新されます。表示言語が英語以外に設定されている Windows デバイスのサポートが追加されています。また、特定のシナリオでプロビジョニングした後、コアデバイスがステータスを AWS IoT Greengrass クラウドサービスに報告しない問題も修正されています。	2022 年 4 月 6 日
AWS IoT Greengrass Core v2.5.4 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.5.4 が提供され、AWSが提供するコンポーネントが更新されます。バグ修正と改良が含まれます。	2022 年 3 月 23 日
AWS IoT Device Tester プログラムによるダウンロード	の IDT を AWS IoT Greengrass V2 プログラムでダウンロードする方法についての情報を追加しました。	2022 年 3 月 15 日

Greengrass Development Kit CLI v1.1.0	Greengrass Development Kit CLI のバージョン 1.1.0 を利用できます。このバージョンでは、`component init` および `component publish` コマンドに新しい引数が追加されています。このバージョンでは、`component publish` コマンドも更新されており、コンポーネントが構築されていない場合にコンポーネントを構築します。	2022 年 2 月 24 日
Shadow Manager v2.1.0 をリリース	Shadow Manager コンポーネント v2.1.0 を利用できます。このリリースでは、コンポーネントがシャドウを と同期する間隔を設定するオプションが追加されています AWS IoT Core。例えば、帯域幅の使用を減らして料金を削減したい場合には、長めの間隔に指定できます。	2022 年 2 月 3 日
AWS IoT Greengrass Core ソフトウェア v2.5.3 用の Dockerfile および Docker イメージ	AWS IoT Greengrass Core ソフトウェア v2.5.3 用の Dockerfile と Docker イメージが利用可能になりました。	2022 年 1 月 12 日

AWS IoT Device Tester v4.5.1 と GGV2Q v2.3.0 のリリース	IDT for AWS IoT Greengrass V2 のバージョン 4.5.1 が利用可能になりました。このリリースには AWS IoT Greengrass V2 認定スイート (GGV2Q) v2.3.0 が含まれており、ハードウェアセキュリティモジュール (HSM) を使用して AWS IoT Greengrass Core ソフトウェアで使用されるプライベートキーと証明書を保存する Linux ベースのデバイスの検証と認定をサポートしています。	2022 年 1 月 11 日
AWS IoT Greengrass Core v2.5.3 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.5.3 が提供され、AWSが提供するコンポーネントが更新されます。これには、ハードウェアセキュリティモジュール (HSM) に安全に保存されているプライベートキーと証明書を使用するように AWS IoT Greengrass Core ソフトウェアを設定するサポートが含まれます。	2022 年 1 月 6 日
AWS IoT Greengrass Core ソフトウェア v2.5.2 の Dockerfile および Docker イメージ	AWS IoT Greengrass Core ソフトウェア v2.5.2 用の Dockerfile と Docker イメージが利用可能になりました。	2021 年 12 月 20 日

AWS IoT Device Tester v4.4.1 と GGV2Q v2.2.1 のリリース	IDT for AWS IoT Greengrass V2 のバージョン 4.4.1 が利用可能になりました。このリリースには AWS IoT Greengrass V2 認定スイート (GGV2Q) v2.2.1 が含まれており、デバイス認定のために Greengrass nucleus バージョン 2.5.2 をサポートしています。	2021 年 12 月 12 日
Amazon Lookout for Vision を使用して機械学習推論を実行する	Greengrass コアデバイスで Lookout for Vision を使用して機械学習推論を実行する方法についての情報を追加しました。Lookout for Vision は、コンピュータビジョンを使用して工業製品にある視覚的な欠陥を検出します。	2021 年 12 月 8 日
AWS IoT Device Tester v4.4.1 と GGV2Q v2.2.0 のリリース	IDT for AWS IoT Greengrass V2 のバージョン 4.4.1 が利用可能になりました。このリリースには AWS IoT Greengrass V2 認定スイート (GGV2Q) v2.2.0 が含まれており、デバイス認定のために Greengrass nucleus バージョン 2.5.2 をサポートしています。	2021 年 12 月 6 日

AWS IoT Greengrass Core v2.5.2 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.5.2 が提供され、AWSが提供するコンポーネントが更新されます。Greengrass nucleus の更新後に発生する Windows サービスの問題を修正しています。また、Windows デバイス上の AWS IoT Device Defender コンポーネントのサポートも含まれています。	2021 年 12 月 3 日
Kinesis Video Streams コンポーネント向けの新しいエッジコネクタ	Kinesis Video Streams コンポーネントのエッジコネクタのバージョン 1.0.0 を利用できます。これは AWSから提供されており、ローカルカメラからビデオフィードを読み取り、ストリームを Kinesis Video Streams にパブリッシュします。このコンポーネントは と統合されており AWS IoT TwinMaker 、Grafana ダッシュボードでビデオストリームやその他のデータを表示および管理できます。	2021 年 11 月 30 日

で Greengrass コアデバイスを管理する AWS Systems Manager	で Greengrass コアデバイスを管理する方法に関する情報を追加しました AWS Systems Manager。Systems Manager は運用データの表示、運用タスクの自動化、セキュリティとコンプライアンスの維持を可能にする AWS サービスです。	2021 年 11 月 29 日
Greengrass Development Kit CLI	AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) に関する情報を追加しました。これは、カスタム Greengrass コンポーネントの開発に役立つローカル開発コンピュータにダウンロードできるツールです。GDK CLI を使用して、カスタムコンポーネントを作成、ビルドおよびパブリッシュできます。	2021 年 11 月 29 日
コミュニティから提供される Greengrass コンポーネント	Greengrass ソフトウェアカタログに関する情報を追加しました。Greengrass コミュニティによって開発された Greengrass コンポーネントのインデックスです。このカタログから、コンポーネントをダウンロード、変更、デプロイして Greengrass アプリケーションを作成できます。	2021 年 11 月 29 日

AWS IoT Greengrass Core v2.5.1 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.5.1 が提供され、AWSが提供するコンポーネントが更新されます。Windows デバイス上の 32 ビット Java へのサポートが含まれています。また、新しいモノグループの削除動作や Windows デバイスでのシステム環境変数の読み込みに関する問題も修正されています。	2021 年 11 月 23 日
AWS IoT Device Tester v4.4.0 と GGV2Q v2.1.0 のリリース	IDT for AWS IoT Greengrass V2 のバージョン 4.4.0 が利用可能になりました。このリリースには AWS IoT Greengrass V2 認定スイート (GGV2Q) v2.1.0 が含まれており、Greengrass nucleus バージョン 2.5.0 を実行する Windows ベースの Greengrass デバイスの認定をサポートしています。	2021 年 11 月 19 日

AWS IoT Greengrass Core v2.5.0 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.5.0 が提供され、AWSが提供するコンポーネントが更新されます。これには、Windows デバイスでの AWS IoT Greengrass Core ソフトウェアの実行のサポートが含まれます。また、モノグループの削除動作が変更され、HTTPS プロキシへのサポートが追加されています。	2021 年 11 月 12 日
SageMaker Edge Manager v1.1.0 のリリース	Amazon SageMaker Edge Manager コンポーネント v1.1.0 が利用可能になりました。このリリースでは、Amazon Linux 2 を実行する Greengrass コアデバイスのサポートが追加されており、デバイス上のキャプチャデータフォルダの場所を指定する新しい設定パラメータが追加されています。	2021 年 11 月 3 日
サービス間の混乱した代理防止の更新	AWS IoT Greengrass V2 では、IAM リソースポリシーで `aws:SourceArn` および `aws:SourceAccount` グローバル条件コンテキストキーを使用して、混乱した代理問題を防止できます。	2021 年 11 月 1 日

クライアントデバイスのコンポーネントの更新	クライアントデバイス認証 v2.0.3、IP ディテクター v2.1.0、MQTT ブリッジ v2.1.0、およびMQTT ブローカー (Moquette) v2.0.2 を利用できます。このリリースでは、デフォルト以外の MQTT ブローカポートを完全にサポートしており、その他のバグ修正と改善も含まれています。	2021 年 10 月 28 日
Shadow Manager v2.0.4 をリリース	Shadow Manager コンポーネント v2.0.4 を利用できます。このリリースでは、シャドウマネージャーが以前に削除されたシャドウの新しく作成されたバージョンを削除する問題が修正されています。このリリース以降、DeleteThingShadow IPC 操作でシャドウバージョンが増加します。	2021 年 10 月 20 日
ログマネージャー v2.2.0 をリリース	ログマネージャーコンポーネント v2.2.0 を利用できます。ログマネージャーは、設定マップの使用を現在サポートし、コンポーネントログ設定を提供します。	2021 年 10 月 20 日

Lambda manager v2.1.4 をリリース	Lambda manager コンポーネント v2.1.4 を利用できます。このリリースでは、NodeJS ランタイムを使用する Lambda 関数が 1 つのメッセージのみを処理する原因となる問題が修正されています。	2021 年 10 月 20 日
Docker コンテナコンポーネントでプロセス間通信、 AWS 認証情報、ストリームマネージャーを使用する	カスタム Docker コンテナコンポーネントのプロセス間通信 (IPC) 、 AWS 認証情報、およびストリームマネージャーを使用する方法についての情報を追加しました。	2021 年 10 月 19 日
新しい nucleus テレメトリエミッタコンポーネント	nucleus テレメトリエミッタコンポーネントのバージョン 1.0.0 を利用できます。このAWSが提供するコンポーネントは、システムヘルステレメトリデータを収集し、ローカルトピックと AWS IoT Core MQTT トピックに継続的に公開します。	2021 年 9 月 30 日
プロキシまたはファイアウォールを介したデバイストラフィックを許可する	セキュリティ対策としてトラフィックを制限できるように、Greengrass コアデバイスが使用するエンドポイントとポートに関する情報を追加しました。	2021 年 9 月 16 日

AWS IoT Device Tester v4.2.0 と GGV2Q v2.0.1 のリリース	IDT for AWS IoT Greengrass V2 のバージョン 4.2.0 が AWS IoT Greengrass V2 認定スイート (GGV2Q) v2.0.1 で更新されました。このリリースでは、デバイスの認定に Greengrass nucleus バージョン 2.4.0 をサポートします。	2021 年 8 月 31 日
機械学習インストーラコンポーネントを更新	DLR インストーラコンポーネント v1.6.5 および TensorFlow Lite インストーラコンポーネント v2.5.4 を使用できます。これらのコンポーネントバージョンには、デフォルトのインストールスクリプトを無効にできる新しい `UseInstaller` 設定パラメータが含まれています。	2021 年 8 月 30 日
の埋め込み Linux サポート AWS IoT Greengrass	の BitBake レシピ AWS IoT Greengrass V2 は、 の `meta-aws`プロジェクトで利用できます GitHub。この recipe を使用して、Yocto Project を使用する Linux ベースのオペレーティングシステムをカスタム構築できます。	2021 年 8 月 20 日
コードの整合性	が Greengrass コアデバイスが からダウンロードするソフトウェアの整合性 AWS IoT Greengrass V2 を検証する方法に関する情報を追加しました AWS クラウド。	2021 年 8 月 19 日

VPC エンドポイント (AWS PrivateLink)	AWS IoT Greengrass は、AWS IoT Greengrass コントロールプレーンのインターフェイス VPC エンドポイント (AWS PrivateLink) をサポートするようになりました。VPC と AWS IoT Greengrass コントロールプレーンの間にプライベート接続を確立できます。	2021 年 8 月 16 日
ストリームマネージャー v2.0.12 をリリース	ストリームマネージャー v2.0.12 を利用できます。このリリースでは、ストリームマネージャーコンポーネントのバージョン 2.0.7 を、v2.0.8 から v2.0.11 の間のバージョンに更新できなかった問題が修正されています。	2021 年 8 月 10 日
AWS IoT Greengrass Core ソフトウェア v2.4.0 用の Dockerfile および Docker イメージ	AWS IoT Greengrass Core ソフトウェア v2.4.0 用の Dockerfile および Docker イメージが利用可能になりました。	2021 年 8 月 9 日
AWS IoT Greengrass Core v2.4.0 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.4.0 が提供され、AWSが提供するコンポーネントが更新されます。これには、コンポーネントのシステムリソースの制限、コンポーネントの一時停止と再開を行う IPC 操作、およびプラグインのプロビジョニングに対するサポートが含まれます。	2021 年 8 月 3 日

新しい AWS IoT SiteWise コンポーネント	用に AWSが提供する次のコンポーネントが追加されました AWS IoT SiteWise: IoT SiteWise OPC-UA コレクター、IoT SiteWise パブリッシャー 、および IoT SiteWise プロセッサ 。	2021 年 7 月 29 日
AWS IoT Device Tester v4.2.0 と GGV2Q v2.0.0 のリリース	IDT for AWS IoT Greengrass V2 のバージョン 4.2.0 が利用可能になりました。このリリースには AWS IoT Greengrass V2 認定スイート (GGV2Q) v2.0.0 が含まれており、Docker コンポーネント、機械学習、ストリームマネージャーのオプション認定テストのサポートも含まれています。	2021 年 7 月 14 日
AWS IoT GreengrassAWS IoT Device SDK for C++ v2 で Core IPC ライブラリが利用可能に	AWS IoT Device SDK for C++ v2 のバージョン 1.13.0 は AWS IoT Greengrass Core IPC をサポートしているため、 Core ソフトウェアとやり取りするコンポーネントを C++ AWS IoT Greengrass で開発できます。	2021 年 7 月 14 日

SageMaker Edge Manager コンポーネント v1.0.2 のリリース	Amazon SageMaker Edge Manager コンポーネント v1.0.2 が利用可能になりました。このリリースは、コンポーネント ライフサイクルのインストールスクリプトを更新します。コアデバイスには Python 3.6 以降をインストールする必要があり、このコンポーネントをデプロイする前に、デバイスにはお使いの Python バージョン向けの `pip` がインストールされている必要があります。	2021 年 7 月 12 日
AWS IoT Device Tester for AWS IoT Greengrass V2 の更新をサポート	IDT for AWS IoT Greengrass V2 バージョン 4.1.0 では、Greengrass nucleus バージョン 2.3.0 を使用したデバイス認定がサポートされるようになりました。	2021 年 7 月 8 日
AWS IoT Greengrass Core ソフトウェア v2.3.0 の Dockerfile および Docker イメージ	AWS IoT Greengrass Core ソフトウェア v2.3.0 用の Dockerfile と Docker イメージが利用可能になりました。	2021 年 7 月 7 日
AWS マネージドポリシー	の AWS マネージドポリシーに関する情報を追加しました AWS IoT Greengrass。	2021 年 7 月 2 日
新たに推奨される JVM オプション	AWS IoT Greengrass Core ソフトウェアのメモリ割り当てを制御するために推奨される JVM オプションに関する情報を追加しました。	2021 年 6 月 30 日

AWS IoT Greengrass Core v2.3.0 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.3.0 が提供され、AWSが提供するコンポーネントが更新されます。これには、デプロイでの大規模なコンポーネント設定ドキュメントに対するサポートが含まれています。	2021 年 6 月 29 日
AWS IoT Greengrass Core ソフトウェア v2.2.0 の Dockerfile および Docker イメージ	AWS IoT Greengrass Core ソフトウェア v2.2.0 用の Dockerfile と Docker イメージが利用可能になりました。	2021 年 6 月 28 日
AWS IoT Device Tester v4.1.0 と GGV2Q v1.1.1 のリリース	IDT for AWS IoT Greengrass V2 のバージョン 4.1.0 が利用可能になりました。このリリースには AWS IoT Greengrass V2 認定スイート (GGV2Q) v1.1.1 が含まれており、Greengrass nucleus v2.2.0、v2.1.0、v2.0.5 を使用したデバイス認定をサポートしています。	2021 年 6 月 18 日
AWS IoT Greengrass Core v2.2.0 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.2.0 が提供され、AWSが提供するコンポーネントが更新されます。これには、クライアントデバイスのサポートを追加し、ローカルシャドウサービスを追加するためにデプロイできるコンポーネントが含まれています。	2021 年 6 月 18 日

Lambda ランチャー v2.0.6 をリリース	Lambda ランチャーコンポーネントのバージョン 2.0.6 を利用できます。このバージョンには、パフォーマンスの向上とバグ修正が含まれています。	2021 年 6 月 13 日
新しい SageMaker Edge Manager コンポーネントがリリースされました	Amazon SageMaker Edge Manager コンポーネントのバージョン 1.0.0 は、 で使用できます AWS IoT Greengrass。このコンポーネントは、Greengrass コアデバイスに SageMaker Edge Manager エージェントバイナリをインストールします。	2021 年 6 月 10 日
コンポーネントタイプ	AWS IoT Greengrassのコンポーネントタイプに関する情報を追加しました。コンポーネントのタイプによって、AWS IoT Greengrass Core ソフトウェアがコンポーネントを実行する方法は異なります。	2021 年 6 月 3 日

AWS IoT Device Tester v4.0.2 と GGV2Q v1.1.0 のリリース	IDT for AWS IoT Greengrass V2 のバージョン 4.0.2が利用可能になりました。このリリースには AWS IoT Greengrass V2 認定スイート (GGV2Q) v1.1.0 が含まれており、Greengrass CLI v2.1.0 での Greengrass nucleus v2.1.0 を使用したデバイス認定をサポートしています。これには、MQTT と Lambda 向けに新たに必須となるテストグループと、その他のマイナーなバグ修正と改善も含まれています。	2021 年 5 月 5 日
AWS IoT Greengrass Core ソフトウェア v2.1.0 の Dockerfile および Docker イメージ	AWS IoT Greengrass Core ソフトウェア v2.1.0 用の Dockerfile と Docker イメージが利用可能になりました。Docker イメージを使用すると、Amazon Linux 2 を基本オペレーティングシステムとして使用する Docker コンテナで AWS IoT Greengrass Core ソフトウェアを実行できます。	2021 年 4 月 27 日

AWS IoT Greengrass Core v2.1.0 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.1.0 が提供され、AWSが提供するコンポーネントが更新されます。これには、プライベート Amazon ECR リポジトリから Docker イメージをダウンロードするために使用できる新しいコンポーネントと、 TensorFlow Lite を使用して機械学習推論を実行するための新しいサンプルコンポーネントが含まれています。	2021 年 4 月 26 日
Secrets Manager を使用するコンポーネントの例	コアデバイスにデプロイする AWS Secrets Manager シークレットの値を出力するコンポーネントの例を追加しました。	2021 年 4 月 8 日
Greengrass コアデバイスの最小限の AWS IoT ポリシー	コアデバイスの基本的な Greengrass 機能をサポートするのに必要な最小限の権限に関する情報が追加されました。	2021 年 4 月 2 日
IPC イベントストリームへのサブスクライブ	プロセス間通信 (IPC) 操作を使用して、Greengrass コアデバイス上の一連のイベントにサブスクライブする方法についての情報を追加しました。	2021 年 4 月 1 日

AWS IoT Device Tester for の更新をサポート AWS IoT Greengrass	IDT for AWS IoT Greengrass V2 バージョン 4.0.1 では、Greengrass CLI バージョン 2.0.5 での Greengrass nucleus バージョン 2.0.5 の使用がデバイス認定でサポートされるようになりました。	2021 年 3 月 17 日
ストリームマネージャーを使用するカスタムコンポーネントを作成する	データストリームを管理するアプリケーションを開発するために、コンポーネント recipe とアーティファクトを設定する方法についての情報を追加しました。	2021 年 3 月 9 日
AWS IoT Greengrass Core v2.0.5 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.0.5 が提供され、AWSが提供するコンポーネントが更新されます。ネットワークプロキシのサポートに関する問題と、 AWS 中国リージョンの Greengrass データプレーンエンドポイントに関する問題が修正されています。	2021 年 3 月 9 日
コンポーネントの環境変数リファレンス	AWS IoT Greengrass Core ソフトウェアがコンポーネントに設定する環境変数に関する情報を追加しました。これらの環境変数を使用して AWS リージョン、モノの名前と Greengrass nucleus バージョンを取得できます。	2021 年 2 月 23 日

手動インストール	必要な AWS リソースを手動で作成する方法、またはファイアウォールまたはネットワークプロキシの背後にインストールする方法に関する情報を追加しました。手動インストールを使用すると、必要な AWS IoT および IAM リソースを作成するため AWS アカウント、 にリソースを作成するアクセス許可をインストーラに付与する必要はありません。また、ポート 443 で、またはネットワークプロキシ経由で接続するようにデバイスを設定することもできます。	2021 年 2 月 17 日
AWS IoT GreengrassAWS IoT Device SDK for Python v2 での Core IPC ライブラリの更新	AWS IoT Device SDK for Python v2 のバージョン 1.5.4 では、 AWS IoT Greengrass Core IPC サービスへの接続に必要なステップが簡素化されています。	2021 年 2 月 11 日
AWS IoT Device Tester for の更新をサポート AWS IoT Greengrass	IDT for AWS IoT Greengrass V2 バージョン 4.0.1 では、Greengrass CLI バージョン 2.0.4 での Greengrass nucleus バージョン 2.0.4 の使用がデバイス認定でサポートされるようになりました。	2021 年 2 月 5 日

Lambda 関数をインポートするための新しいチュートリアル	Greengrass コアデバイスで実行されるコンポーネントとして Lambda 関数をインポートするための新しいコンソールベースのチュートリアルを追加しました。	2021 年 2 月 5 日
AWS IoT Greengrass Core v2.0.4 ソフトウェア更新	このリリースでは、Greengrass nucleus コンポーネントのバージョン 2.0.4 が提供されています。これには、ポート 443 経由で HTTPS 通信を設定するための新しい `greengrassDataPlanePort` パラメータとバグの修正が含まれています。 AWS IoT Greengrass Core ソフトウェアインストーラを で実行`sts:GetCallerIdentity` するときに、最小限の IAM ポリシーで `iam:GetPolicy` と が必要になりました `--provision true` 。	2021 年 2 月 4 日
新しいセキュアトンネリングコンポーネントのリリース	セキュアトンネリングコンポーネントのバージョン 1.0.0 は、 で使用できます AWS IoT Greengrass。この AWSが提供するコンポーネントは、AWS IoT セキュアトンネリングを使用して、制限されたファイアウォールの背後にある Greengrass コアデバイスとの安全な双方向通信を確立します。	2021 年 1 月 21 日

AWS IoT Device Tester for AWS IoT Greengrass v4.0.1 がリリースされました	IDT for AWS IoT Greengrass V2 のバージョン 4.0.1 が利用可能になりました。このバージョンでは、IDT を使用して、デバイス検証用のカスタムテストスイートを開発および実行できます。これには、macOS および Windows 用のコード署名付き IDT アプリケーションも含まれています。	2020 年 12 月 22 日
の初回リリース AWS IoT Greengrass Version 2	AWS IoT Greengrass V2 は、の新しいメジャーバージョンリリースです AWS IoT Greengrass。このバージョンには、モジュラーソフトウェアコンポーネントや継続的なデプロイなどの複数の機能が追加されています。これらの機能により、エッジアプリケーションの開発と管理が容易になります。	2020 年 12 月 15 日

AWS 用語集

最新の AWS 用語については、 AWS の用語集 リファレンスのAWS 用語集を参照してください。

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。