aws

開発者ガイド

AWS Device Farm

API バージョン 2015-06-23

AWS Device Farm: 開発者ガイド

Table of Contents

AWS Device Farm について

Device Farm は、アマゾン ウェブ サービス (AWS) によりホストされている実際の物理的な電話やタブレットで、Android や iOS、およびウェブアプリケーションをテストしてやり取りできるアプリケーションテストサービスです。

Device Farm を使用する方法は 2 つあります。

- さまざまなテストフレームワークを使用したアプリケーションの自動テスト
- 読み込み、実行、リアルタイムでアプリケーションとやり取り可能なデバイスへのリモートアクセス。

> **Note**
>
> Device Farm は、`us-west-2` (オレゴン) リージョンでのみ使用可能です。

自動アプリケーションテスト

Device Farm により、独自のテストをアップロードしたり、組み込まれているスクリプトフリーの互換性テストを使用できます。テストは並列実行されるため、テストは複数のデバイスで数分のうちに開始されます。

テストが完了すると、ハイレベルの結果、低レベルのログ、ピクセルからピクセルへのスクリーンショット、パフォーマンスデータを含むテストレポートが更新されます。

Device Farm は、ネイティブかつハイブリッドな Android アプリケーション、および iOS アプリケーション、PhoneGap、Titanium、Xamarin、Unity、およびその他のフレームワークで作成されたもののテストをサポートしています。インタラクティブなテスト用に Android アプリケーションおよび iOS アプリケーションのリモートアクセスをサポートしています。サポートされているテストタイプの詳細については、「AWS Device Farm のテストタイプによる作業」を参照してください。

リモートアクセスでの交信

リモートアクセスを使用すると、ウェブブラウザを介してリアルタイムでデバイスのスワイプ、ジェスチャ、および操作を行うことができます。デバイスのリアルタイムでの操作が役立つ状況は数多

くあります。例えば、カスタマーサービス担当者は、デバイスの使用やセットアップを通してお客様に案内することができます。また、特定のデバイスで実行されているアプリケーションの使用を通して、お客様に説明することもできます。リモートアクセスセッションで実行されているデバイスにアプリケーションをインストールでき、お客様の問題や報告されたバグを再現できます。

リモートアクセスセッション中、Device Farm は、デバイスとのやり取りで実行されたアクションの詳細を収集します。セッションの終了時に、これらの詳細を含むログとセッションの動画キャプチャが生成されます。

用語

Device Farm では、情報を整理する方法を定義する、次の用語が導入されます。

デバイスプール

プラットフォーム、製造元、モデルなど、一般的に類似した特性を共有するデバイスのコレクション。

ジョブ

1 つのデバイスに対して単一アプリケーションをテストするための Device Farm へのリクエスト。ジョブは、1 つ以上のスイートで構成されます。

計測

デバイスの請求を指します。ドキュメントおよび API リファレンスの「測定デバイス」または「測定対象外デバイス」へのリファレンスが表示されます。料金の詳細については、「AWS Device Farm 料金表」を参照してください。

プロジェクト

実行を含む論理ワークスペース、1 つ以上のデバイスに対する単一のアプリケーションのテストごとに 1 つずつ実行。プロジェクトを使用すると、選択した任意の方法でワークスペースを整理できます。例えば、アプリケーションのタイトルごとに 1 つのプロジェクトがある場合もあれば、プラットフォームごとに 1 つのプロジェクトがある場合もあります。プロジェクトは必要な数だけ作成できます。

レポート

これには、実行に関する情報、1 つ以上のデバイスに対して単一アプリケーションをテストするための、Device Farm に対するリクエストが含まれます。詳細については、「AWS Device Farm でのレポート」を参照してください。

run

特定の一連のデバイスで実行される、アプリケーションの特定のビルド、特定の一連のテスト。実行によって、結果のレポートが生成されます。実行は、1 つ以上のジョブで構成されます。詳細については、「実行」を参照してください。

セッション

ウェブブラウザを通した実際の物理デバイスとのリアルタイムのやり取りです。詳細については、「セッション」を参照してください。

スイート

テストパッケージ内の階層構造のテストです。スイートは、1 つ以上のテストで構成されます。

テスト

テストパッケージ内の個別のテストケース。

Device Farm の詳細については、「概念」を参照してください。

セットアップ

Device Farm を使用するには、「設定」を参照してください。

AWS Device Farm のセットアップ

Device Farm を初めて使用する場合は、事前に以下のタスクを完了する必要があります:

トピック

ステップ 1: にサインアップする AWS

Amazon Web Services (AWS) にサインアップします。

がない場合は AWS アカウント、次の手順を実行して作成します。

にサインアップするには AWS アカウント

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、電話キーパッドで検証コードを入力するように求められます。

   にサインアップすると AWS アカウント、 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、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。

ステップ 2: AWS アカウントで IAM ユーザーを作成または使用する

Device Farm へのアクセスに AWS ルートアカウントを使用しないことをお勧めします。代わりに、AWS アカウントに (IAM) ユーザーを作成し AWS Identity and Access Management (または既存のユーザーを使用) 、その IAM ユーザーを使用して Device Farm にアクセスします。

詳細については、「IAM ユーザーの作成 (AWS Management Console)」を参照してください。

ステップ 3: IAM ユーザーに Device Farm へアクセスする権限を付与する

IAM ユーザーに Device Farm へアクセスする権限を付与します。そのためには、IAM でアクセスポリシーを作成後、そのアクセスポリシーを以下のように IAM ユーザーに割り当てます。

> **Note**
>
> 次の手順を完了するために使用する AWS ルートアカウントまたは IAM ユーザーには、次の IAM ポリシーを作成して IAM ユーザーにアタッチするアクセス許可が必要です。詳細については、「ポリシーによる作業」を参照してください。

1. 次の JSON 本文を使用してポリシーを作成します。などのわかりやすいタイトルを付けます*DeviceFarmAdmin*。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "devicefarm:*"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

   IAM ポリシーの作成の詳細については、「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。

2. 作成した IAM ポリシーを新規ユーザーにアタッチします。ユーザーに IAM ポリシーをアタッチする方法の詳細については、「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。

ポリシーをアタッチすることで、IAM ユーザーには、その IAM ユーザーに関連付けられたすべての Device Farm アクションとリソースへのアクセス権が付与されます。限定された一連の Device Farm アクションとリソースに IAM ユーザーを制限する方法については、「AWS Device Farm のアイデンティティとアクセス管理」を参照してください。

次のステップ

これで Device Farm の使用を開始する準備ができました。Device Farm の開始 を参照してください。

Device Farm の開始

ここでは、Device Farm を使用してネイティブ Android または iOS アプリケーションをテストする方法を説明します。Device Farm コンソールを使えば、プロジェクトの作成、.apk または .ipa ファイルのアップロード、一連の標準テストの実行を行い、その結果を表示できます。

> **Note**
>
> Device Farm は `us-west-2` (オレゴン) AWS リージョンでのみ使用可能です。

トピック

前提条件

作業を開始する前に、次の要件を満たしていることを確認してください:

- 設定 の各ステップを完了します。Device Farm にアクセスするには、AWS アカウント、および権限を有する AWS Identity and Access Management (IAM) ユーザーが必要です。

- Android の場合は、.apk (Android アプリケーションパッケージ) ファイルが必要です。iOS の場合は、.ipa (iOS アプリアーカイブ) ファイルが必要です。ここでは、後で同ファイルを Device Farm にアップロードします。

  > **Note**
  >
  > .ipa ファイルがシミュレーター用ではなく iOS デバイス用に作成されていることを確認します。

- (オプション) Device Farm がサポートするいずれかのテストフレームワークからの 1 つのテストが必要です。そのテストパッケージを Device Farm にアップロードし、後でテストを実行します。使用可能なテストパッケージがない場合、標準ビルトインテストスイートを指定し、実行できます。詳細については、「AWS Device Farm のテストタイプによる作業」を参照してください。

ステップ 1: コンソールにサインインする

Device Farm コンソールを使用して、テスト用のプロジェクトや実行を作成および管理できます。プロジェクトや実行については、この後半で説明します。

- https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。

ステップ 2: プロジェクトを作成する

Device Farm でアプリケーションをテストするには、まずプロジェクトを作成する必要があります。

1. ナビゲーションペインで、 [モバイルデバイスのテスト] を選択し、そして次に、[プロジェクト] を選択します
2. [モバイルデバイステストプロジェクト] で、[新規プロジェクト] を選択します。
3. [プロジェクトを作成] で、[プロジェクト名] (例えば、**MyDemoProject**) を入力します。
4. [作成] を選択します。

   コンソールが、新しく作成したプロジェクトの「自動テスト」ページを開きます。

ステップ 3: 実行を作成し開始する

プロジェクトが生まれ、実行を作成して開始できます。詳細については、「実行」を参照してください。

1. 「[自動テスト」ページで、[新規実行を作成] を選択します。
2. 「アプリケーションを選択する」ページで、[モバイルアプリケーション] 下にある [ファイルを選択] を選択して、コンピュータから Android (.apk) または iOS (.ipa) ファイルを選択します。または、コンピュータからファイルをドラッグしてコンソールにドロップします。
3. **my first test** などの [実行名] を入力します。デフォルトで、Device Farm コンソールはファイル名を使用します。

4. [次へ] を選択します。
5. 「構成する」ページで、[テストフレームワークをセットアップ] 下にあるテストフレームワークかビルトインテストスイートのうち 1 つを選択します。各オプションの詳細については、「テストタイプとフレームワーク」を参照してください。
   - Device Farm のテストをまだパッケージしていない場合は、[ビルトイン: ファズ] を選択して、標準ビルトインテストスイートを実行します。[イベント数]、[イベントスロットル]、および [乱数シード] にはデフォルト値をそのまま使用できます。詳細については、「the section called “ビルトイン: ファズ (Android および iOS)”」を参照してください。
   - サポートされているテストフレームワークのいずれかのテストパッケージがある場合は、対応するテストフレームワークを選択し、テストを含むファイルをアップロードします。
6. [次へ] を選択します。
7. 「デバイスを選択する」ページの、[デバイスプール] で、[上位デバイス] を選択します。
8. [次へ] を選択します。
9. 「デバイス状態を指定する」ページで、次のいずれかを実行します:
   - Device Farm が実行中に使用する追加データを提供するには、[データを追加] で、.zip ファイルをアップロードします。
   - 実行のために他のアプリケーションをインストールするには、[他のアプリケーションをインストール] で、アプリケーションの .apk または .ipa ファイルをアップロードします。インストール順序を変更するには、ファイルをドラッグアンドドロップします。
   - 実行のために Wi-Fi、Bluetooth、GPS、または NFC 無線を作動させるには、[無線状態を設定] で、該当するチェックボックスをオンにします。

     > **Note**
     >
     > デバイスの無線状態の設定は、現時点では Android ネイティブテストにのみ使用できます。

   - 実行中に場所固有の動作をテストするには、[デバイスの場所] で、プリセットの [緯度] および [経度] 座標を指定します。
   - 実行用のデバイスの言語とリージョンをプリセットするには、[デバイスロケール] で、ロケールを選択します。
   - 実行のためにネットワークプロファイルをプリセットするには、[ネットワークプロファイル] で、キュレーションされたプロファイルを選択します。または、[ネットワークプロファイルを作成] を選択して、独自のものを作成します。

10. [次へ] を選択します。
11. 「実行を確認して開始する」ページで、[実行を確認して開始] を選択します。

Device Farm は、デバイスが利用可能になると、通常数分以内に実行を開始します。実行ステータスを表示するには、プロジェクトの「自動テスト」ページで、実行の名前を選択します。「実行する」ページで、[デバイス] のデバイステーブルでは、最初に保留中アイコン

が表示され、テストが始まると実行中アイコン

に切り替わります。各テストが終了すると、コンソールでは、テスト結果アイコンがデバイス名の横に表示されます。すべてのテストが完了すると、実行の横にある保留中アイコンがテスト結果アイコンに変わります。

ステップ 4: 実行の結果を表示する

実行からテスト結果を表示するには、プロジェクトの「自動テスト」ページで、実行の名前を選択します。概要ページが表示されます:

- 結果ごとのテストの合計数。
- 特別な警告または失敗があるテストのリスト。
- それぞれにテスト結果が付いたデバイスのリスト。
- 実行時にキャプチャされたスクリーンショット、デバイス別にグループ化。
- 解析結果をダウンロードするためのセクション。

詳細については、「Device Farm でのテストレポートによる作業」を参照してください。

次のステップ

Device Farm の詳細については、「概念」を参照してください。

Device Farm でデバイススロットを購入する

Device Farm コンソール、 AWS Command Line Interface （ AWS CLI ）、または Device Farm API を使用して、デバイススロットを購入できます。

トピック

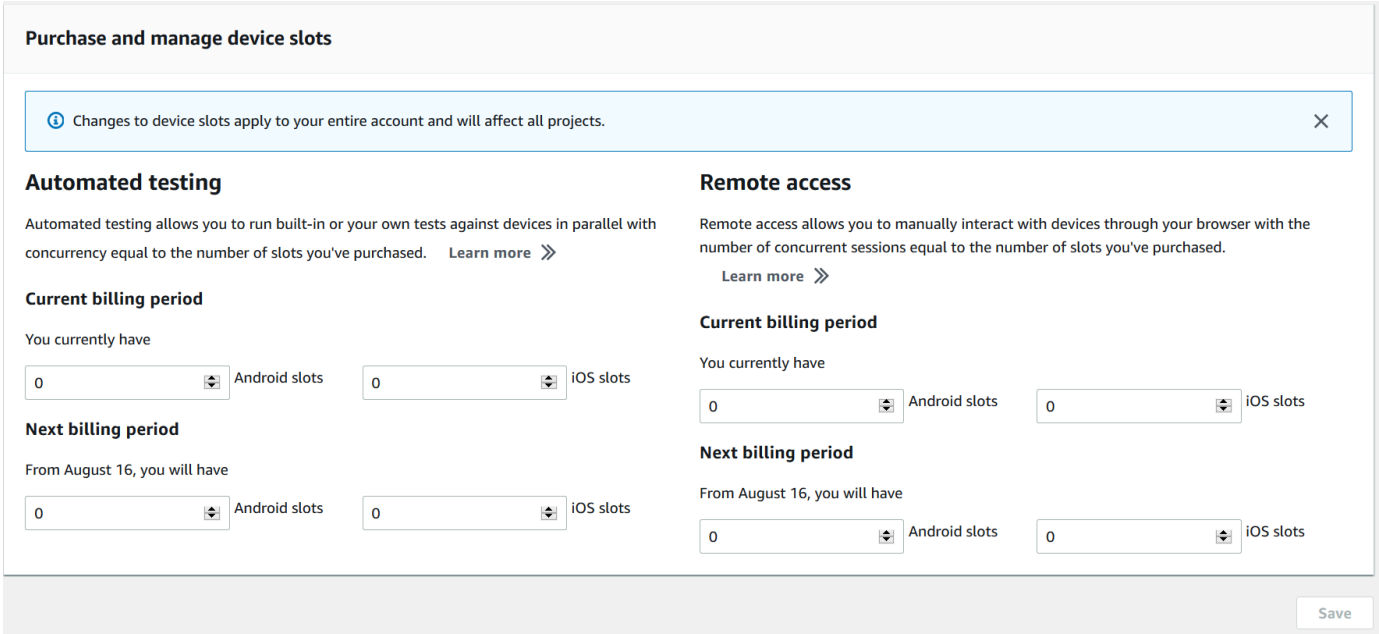
デバイススロットを購入する (コンソール)

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. ナビゲーションペインにある、 [モバイルデバイスのテスト] を選択し、[デバイススロット] を選択します。
3. 「デバイススロットを購入および管理する」ページでは、購入する自動テストデバイスおよびリモートアクセスデバイスのスロットの数を選択して、独自のカスタムパッケージを作成できます。現在および次の請求期間の両方にスロットの量を指定します。

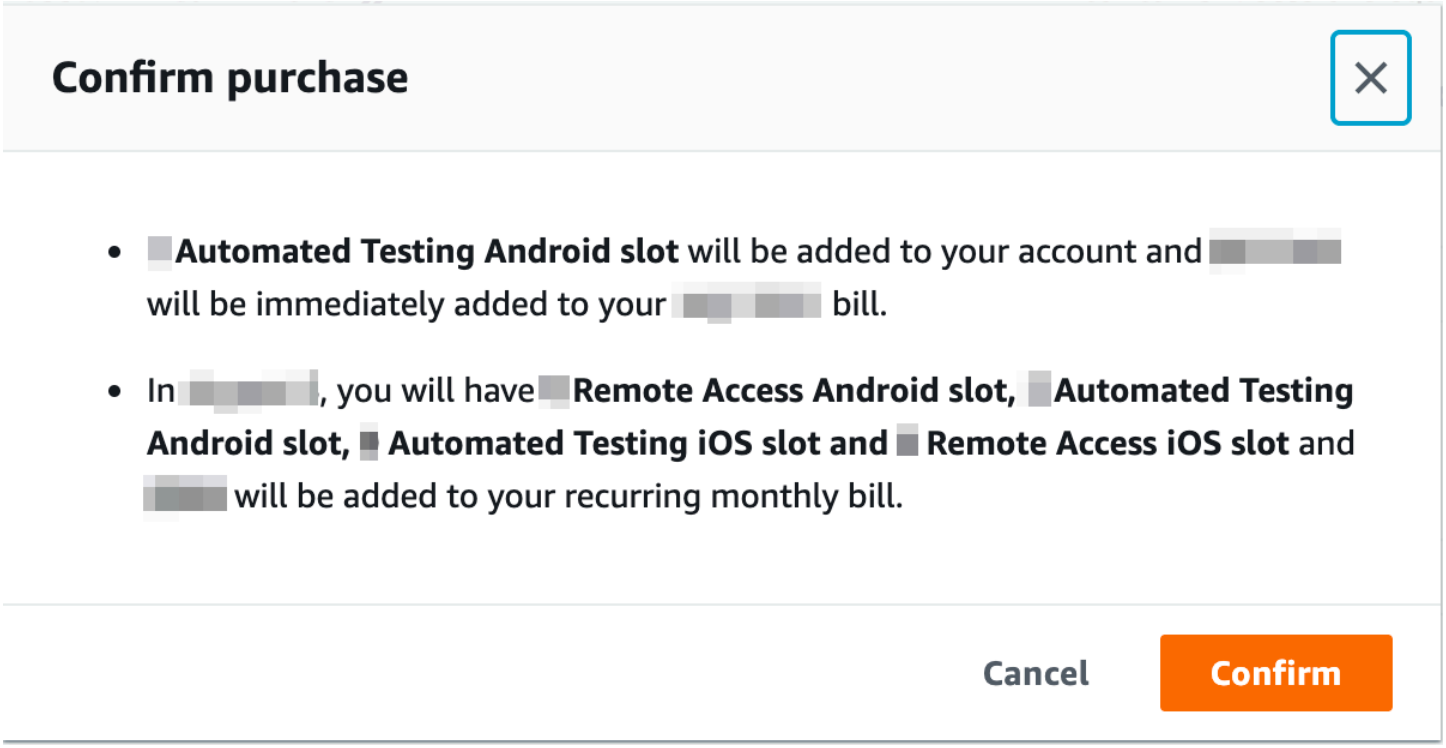
   スロットの量を変更すると、テキストが、請求金額によって動的に更新します。詳細については、「AWS Device Farm の料金」を参照してください。

   > **Important**
   >
   > デバイススロットの数を変更しても、お問い合わせまたは購入メッセージが表示される場合、 AWS アカウントはリクエストしたデバイススロットの数の購入をまだ承認されていません。
   > そこでは、Device Farm サポートチームに E メールを送信するように求められます。E メールには、購入したい各デバイスタイプの数、および請求サイクルを指定します。

> **Note**
>
> デバイススロットを変更すると、アカウント全体に適用され、すべてのプロジェクトに影響します。



4. [購入] を選択します。購入の確認ウィンドウが表示されます。情報を確認し、確認を選択してトランザクションを完了します。



「デバイススロットを購入および管理する」ページで、現在所有するデバイススロットの数を確認できます。スロット数が増減する場合は、変更を行った日付から 1 か月後に保有することになるスロットの数が表示されます。

デバイススロットを購入する (AWS CLI)

purchase-offering コマンドを実行して購入ができます。

購入できるデバイススロットの最大数や、無料試用の残り分数を含む Device Farm アカウント設定を一覧表示するには、get-account-settings コマンドを実行します。結果は次のように表示されます:

```
{
    "accountSettings": {
        "maxSlots": {
            "GUID": 1,
            "GUID": 1,
            "GUID": 1,
            "GUID": 1
        },
        "unmeteredRemoteAccessDevices": {
            "ANDROID": 0,
            "IOS": 0
```

```
        },
        "maxJobTimeoutMinutes": 150,
        "trialMinutes": {
            "total": 1000.0,
            "remaining": 954.1
        },
        "defaultJobTimeoutMinutes": 150,
        "awsAccountNumber": "AWS-ACCOUNT-NUMBER",
        "unmeteredDevices": {
            "ANDROID": 0,
            "IOS": 0
        }
    }
}
```

利用可能なデバイススロット商品を一覧表示するには、list-offerings コマンドを実行します。結果は次のように表示されます:

```
{
    "offerings": [
        {
            "recurringCharges": [
                {
                    "cost": {
                        "amount": 250.0,
                        "currencyCode": "USD"
                    },
                    "frequency": "MONTHLY"
                }
            ],
            "platform": "IOS",
            "type": "RECURRING",
            "id": "GUID",
            "description": "iOS Unmetered Device Slot"
        },
        {
            "recurringCharges": [
                {
                    "cost": {
                        "amount": 250.0,
                        "currencyCode": "USD"
                    },
                    "frequency": "MONTHLY"
```

```
                }
            ],
            "platform": "ANDROID",
            "type": "RECURRING",
            "id": "GUID",
            "description": "Android Unmetered Device Slot"
        },
        {
            "recurringCharges": [
                {
                    "cost": {
                        "amount": 250.0,
                        "currencyCode": "USD"
                    },
                    "frequency": "MONTHLY"
                }
            ],
            "platform": "ANDROID",
            "type": "RECURRING",
            "id": "GUID",
            "description": "Android Remote Access Unmetered Device Slot"
        },
        {
            "recurringCharges": [
                {
                    "cost": {
                        "amount": 250.0,
                        "currencyCode": "USD"
                    },
                    "frequency": "MONTHLY"
                }
            ],
            "platform": "IOS",
            "type": "RECURRING",
            "id": "GUID",
            "description": "iOS Remote Access Unmetered Device Slot"
        }
    ]
}
```

利用可能な売り出しプロモーションを一覧表示するには、list-offering-promotions コマンドを実行します。

> **Note**
>
> このコマンドは、未購入のプロモーションのみ戻します。あるプロモーションの売り出しで 1 つ以上のスロットを購入すると、そのプロモーションは結果に表示されなくなります。

結果は次のように表示されます:

```
{
    "offeringPromotions": [
        {
            "id": "2FREEMONTHS",
            "description": "New device slot customers get 3 months for the price of 1."
        }
    ]
}
```

売り出しステータスを取得するには、get-offering-status コマンドを実行します。結果は次のように表示されます:

```
{
    "current": {
        "GUID": {
            "offering": {
                "platform": "IOS",
                "type": "RECURRING",
                "id": "GUID",
                "description": "iOS Unmetered Device Slot"
            },
            "quantity": 1
        },
        "GUID": {
            "offering": {
                "platform": "ANDROID",
                "type": "RECURRING",
                "id": "GUID",
                "description": "Android Unmetered Device Slot"
            },
            "quantity": 1
        }
    },
    "nextPeriod": {
```

```
        "GUID": {
            "effectiveOn": 1459468800.0,
            "offering": {
                "platform": "IOS",
                "type": "RECURRING",
                "id": "GUID",
                "description": "iOS Unmetered Device Slot"
            },
            "quantity": 1
        },
        "GUID": {
            "effectiveOn": 1459468800.0,
            "offering": {
                "platform": "ANDROID",
                "type": "RECURRING",
                "id": "GUID",
                "description": "Android Unmetered Device Slot"
            },
            "quantity": 1
        }
    }
}
```

renew-offering および list-offering-transactions のコマンドはこの機能についても使用できます。詳細については「AWS CLI リファレンス」を参照してください。

デバイススロットを購入する (API)

1. GetAccount設定オペレーションを呼び出して、アカウント設定を一覧表示します。
2. ListOfferings オペレーションを呼び出して、使用可能なデバイススロットサービスを一覧表示します。
3. ListOfferingプロモーションオペレーションを呼び出して、利用可能なオファープロモーションを一覧表示します。

> **Note**
>
> このコマンドは、未購入のプロモーションのみ戻します。ある売り出しプロモーションで 1 つ以上のスロットを購入すると、そのプロモーションは結果に表示されなくなります。

4. オファリングを購入するには、 PurchaseOfferingオペレーションを呼び出します。
5. GetOfferingステータスオペレーションを呼び出して、提供ステータスを取得します。

RenewOffering および ListOfferingトランザクションコマンドは、この機能でも使用できます。

Device Farm API の使用についての情報は、「Device Farm の自動化」を参照してください。

デバイススロットをキャンセルする (コンソール)

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. ナビゲーションペインにある、 [モバイルデバイスのテスト] を選択し、[デバイススロット] を選択します。
3. デバイススロットの購入と管理ページで、次の請求期間 の 値を減らすことで、自動テストとリモートアクセスの両方のデバイススロットの数を減らすことができます。次の請求サイクルでアカウントに請求される金額は、請求期間フィールドの下に表示されます。
4. [保存] を選択します。変更の確認ウィンドウが表示されます。情報を確認し、確認を選択してトランザクションを完了します。

デバイススロットをキャンセルする (AWS CLI)

renew-offering コマンドを実行して、次の請求サイクルのデバイス数を変更できます。

デバイススロットをキャンセルする (API)

RenewOffering オペレーションを呼び出して、アカウント内のデバイスの数を変更します。

AWS Device Farm の概念

このセクションでは、Device Farm の重要な概念について説明します。

- AWS Device Farm でのデバイスサポート
- テスト環境
- 実行
- アプリケーション
- AWS Device Farm でのレポート
- セッション

Device Farm でサポートされているテストタイプの詳細については、「AWS Device Farm のテストタイプによる作業」を参照してください。

AWS Device Farm でのデバイスサポート

以下のセクションでは、Device Farm でのデバイスサポート情報を伝えます。

トピック

- サポートされるデバイス
- デバイスプール
- プライベートデバイス
- デバイスのブランディング
- デバイススロット
- プリインストールされたデバイスアプリケーション
- デバイス機能

サポートされるデバイス

Device Farm は、Android とiOS の人気デバイスおよびオペレーティングシステムの何百種類もの組み合わせをサポートします。利用可能デバイスのリストは、新しいデバイスが市場に参入するにつれて大きくなります。デバイスの完全リストは、「デバイスリスト」で参照できます。

デバイスプール

Device Farm は、デバイスをテスト用デバイスプールの中に編成します。これらのデバイスプールには、Android または iOS でのみ実行されるデバイスなど、関連するデバイスが含まれます。Device Farm は、最上位デバイス用など、精選されたデバイスプールを提供します。パブリックデバイスとプライベートデバイスを混合したデバイスプールも作成できます。

プライベートデバイス

プライベートデバイスでは、テストニーズに応じてハードウェアとソフトウェアの構成を細かく指定できます。ルート権限を取得した Android デバイスなどの特定構成は、プライベートデバイスとしてサポートできます。各プライベートデバイスは、Amazon データセンターで Device Farm がユーザーに代わってデプロイする物理デバイスです。プライベートデバイスは自分専用として、自動テストと手動テストの両方に使用できます。サブスクリプション終了を選択すると、ご使用の環境からハードウェアが削除されます。詳細については、「プライベートデバイス」および「AWS Device Farm でのプライベートデバイスによる作業」を参照してください。

デバイスのブランディング

Device Farm は、さまざまな OEM の物理的なモバイルデバイスとタブレットデバイスでテストを実行します。

デバイススロット

デバイススロットは同時実行に対応し、そこでは購入したデバイススロットの数によってテストまたはリモートアクセスセッションで実行できるデバイスの数が決まります。

デバイススロットには、次の 2 種類があります:

- リモートアクセスデバイススロットはリモートアクセスセッションで同時実行できます。

  1 つのリモートアクセスデバイススロットがある場合は、一度に 1 つのリモートアクセスセッションしか実行できません。追加のリモートテストデバイススロットを購入すると、複数セッションを同時実行できます。
- 自動テストデバイススロットは、テストを同時実行できます。

  自動テストデバイススロットが 1 つの場合、テストは一度に 1 つのデバイスでのみ実行できます。追加の自動テストデバイススロットを購入すると、複数デバイスで複数テストを同時実行し、テスト結果を迅速に取得できます。

デバイススロットはデバイスファミリー (自動テスト用の Android または iOS デバイス、および、リモートアクセス用の Android または iOS デバイス) に基づいて購入できます。詳細については、「Device Farm 料金表」を参照してください。

プリインストールされたデバイスアプリケーション

Device Farm のデバイスには、メーカーやキャリアによって既にインストールされているアプリケーションがいくつか含まれています。

デバイス機能

デバイスはいずれも、インターネットに Wi-Fi 接続できます。それらにはキャリア接続がなく、電話をかけたり SMS メッセージを送信することはできません。

前面または背面のカメラをサポートする任意のデバイスで写真を撮ることができます。デバイスのマウント方法により、写真が暗く、ぼやける場合があります。

Google Play サービスはサポートされているデバイスにはインストールされていますが、これらのデバイスには有効な Google アカウントはありません。

AWS Device Farm でのテスト環境

AWS Device Farm では、自動化テストの実行用にカスタムテスト環境と標準テスト環境の両方が用意されています。自動化テストを完全にコントロールするには、カスタムのテスト環境を選択します。または、Device Farm のデフォルト標準テスト環境を選択できます。このテスト環境では、自動化テストスイートの各テストが細かくレポートされます。

トピック

- 標準テスト環境
- カスタムテスト環境

標準テスト環境

標準環境でテストを実行する場合、Device Farm では、テストスイートのすべてのケースで詳細なログまたはレポートが提供されます。各テストのパフォーマンスデータ、動画、スクリーンショット、ログを表示して、アプリケーションでの問題の特定や修正ができます。

> **Note**
>
> Device Farm は、標準環境で細かいレポートを提供するため、テスト実行時間は、ローカルでテストを実行する場合よりも長くなる場合があります。実行時間を早めるには、カスタムテスト環境でテストを実行します。

カスタムテスト環境

テスト環境のカスタマイズにより、Device Farm が実行するコマンドを指定してテストを実行できます。これにより確実に、ローカルマシン上でテストを実行した場合と同様に、Device Farm のテストを実行できます。このモードでテストを実行して、テストのライブログおよびビデオストリーミングを有効にすることもできます。カスタマイズされたテスト環境でテストを実行する場合、各テストケースの細かいレポートは取得できません。詳細については、「カスタムテスト環境による作業」を参照してください。

オプションでカスタムテスト環境を使用でき、Device Farm コンソール、 AWS CLI、または Device Farm API によりテスト実行を作成できます。

詳細については、「AWS CLIを使用したカスタムテスト仕様のアップロード」および「Device Farm でのテスト実行の作成」を参照してください。

AWS Device Farm での実行

次のセクションには、Device Farm での実行に関する情報が含まれています。

Device Farm での実行では、特定の一連のテストで、特定の一連のデバイスで実行される、アプリケーションの特定のビルドが表されます。実行では、実行の結果に関する情報を含むレポートが作成されます。実行には、1 つ以上のジョブが含まれます。

トピック

- 構成を実行する
- ファイル保持を実行する
- デバイス状態を実行する
- 並列実行
- 実行タイムアウトの設定
- 実行での広告

- 実行でのメディア
- 実行のための一般的なタスク

構成を実行する

実行の一部として、Device Farm が現在のデバイス設定を上書きする設定を加えられます。これには、緯度と経度の座標、ロケール、無線状態 (Bluetooth、GPS、NFC、Wi-Fi など)、追加データ (.zip ファイル内)、補助アプリケーション (テスト前にインストールが必要) が含まれます。

ファイル保持を実行する

Device Farm は、アプリケーションやファイルを 30 日間保存し、その後システムから削除します。ただし、ファイルはいつでも削除できます。

Device Farm は、実行結果、ログ、およびスクリーンショットを 400 日間保存し、その後システムから削除します。

デバイス状態を実行する

Device Farm は、次のジョブに使用可能とするため、いつでもデバイスを再起動します。

並列実行

Device Farm はデバイスが使用可能になると同時にテストを実行します。

実行タイムアウトの設定

各デバイスでテスト実行を停止するまでのテスト実行時間を設定できます。例えば、テスト完了までにデバイスあたり 20 分かかる場合、デバイスあたり 30 分のタイムアウトを選択する必要があります。

詳細については、「AWS Device Farm でのテスト実行の実行タイムアウトを設定する」を参照してください。

実行での広告

Device Farm にアップロードする前に、アプリケーションから広告を削除することをおすすめします。実行中に広告が表示されることは保証できません。

実行でのメディア

アプリケーションに付随するメディアやその他のデータを提供できます。追加データは、4 GB 以下のサイズの .zip ファイルで提供する必要があります。

実行のための一般的なタスク

詳細については、「Device Farm でのテスト実行の作成」および「AWS Device Farm でのテスト実行による作業」を参照してください。

AWS Device Farm のアプリケーション

以下のセクションには、Device Farm のアプリケーション動作に関する情報が含まれています。

トピック

- 計測アプリケーション
- 実行するアプリケーションの再署名
- 実行での難読化アプリケーション

計測アプリケーション

アプリケーションを計測したり、アプリケーションのソースコードを Device Farm に提供する必要はありません。Android アプリケーションは変更なしで送信できます。iOS アプリケーションは、シミュレータではなく、iOS デバイスターゲットで作成する必要があります。

実行するアプリケーションの再署名

iOS アプリケーションの場合、プロビジョニングプロファイルに Device Farm UUID を追加する必要はありません。Device Farm は、組み込みプロビジョニングプロファイルをワイルドカードプロファイルに置き換え、アプリケーションに再署名します。補助データを提供する場合、Device Farm では、Device Farm がインストールする前にそれをアプリケーションのパッケージに追加し、その補助データがアプリケーションのサンドボックスに存在するようにします。アプリに再署名すると、アプリグループ、関連ドメイン、ゲームセンター、 HealthKit、ワイヤレスアクセサリ設定 HomeKit、アプリ内購入、アプリ間オーディオ、Apple Pay、プッシュ通知、VPN 設定とコントロールなどの使用権限が削除されます。

Android アプリケーションの場合、Device Farm がアプリケーションに再署名します。これにより、Google Maps Android API など、アプリの署名に依存する機能が破損したり、 などの製品から著作権侵害や改ざん防止の検出がトリガーされたりする可能性があります DexGuard。

実行での難読化アプリケーション

Android アプリの場合、アプリが難読化されても、 を使用している場合は Device Farm でテストできます ProGuard。ただし、著作権侵害対策 DexGuard で を使用すると、Device Farm はアプリケーションに対して再署名してテストを実行できません。

AWS Device Farm でのレポート

以下のセクションでは、Device Farm テストレポートについて説明します。

トピック

- レポート保持
- レポート内容
- レポートのログ
- レポート用の一般的タスク

レポート保持

Device Farm は 400 日間レポートを保存します。これらのレポートには、メタデータ、ログ、スクリーンショット、パフォーマンスデータが含まれます。

レポート内容

Device Farm のレポートには、成功および失敗情報、クラッシュレポート、テストとデバイスのログ、スクリーンショット、およびパフォーマンスデータが含まれています。

レポートには、デバイス別詳細データとハイレベル結果が含まれます (例: 特定問題の発生回数)。

レポートのログ

レポートには、Android テストの完全な logcat キャプチャ、および iOS テストの完全なデバイスコンソールログが含まれます。

レポート用の一般的タスク

詳細については、「Device Farm でのテストレポートによる作業」を参照してください。

AWS Device Farm でのセッション

Device Farm を使えば、ウェブブラウザのリモートアクセスセッションにより、Android および iOS のアプリケーションのインタラクティブテストを実行できます。このようなインタラクティブテストは、問題を訴える顧客からの電話にサポートエンジニアが順を追った対応するのに役立ちます。開発者は問題の原因を分離するために、特定のデバイスで問題を再現できます。リモートセッションを使用して、対象の顧客と共にユーザビリティテストを実施できます。

トピック

リモートアクセスでサポートされるデバイス

Device Farm は、特別および一般的な、多数の Android および iOS デバイスをサポートします。利用可能デバイスのリストは、新しいデバイスが市場に参入するにつれて拡大します。Device Farm コンソールには、リモートアクセスで利用可能な Android および iOS デバイスの最新リストが表示されます。詳細については「AWS Device Farm でのデバイスサポート」を参照してください。

セッションファイルの保持

Device Farm は、アプリケーションやファイルを 30 日間保存し、その後システムから削除します。ただし、お客様はいつでもファイルを削除できます。

Device Farm は、セッションログとキャプチャしたビデオを 400 日間保存し、その後システムから削除します。

計測アプリケーション

アプリケーションを計測したり、アプリケーションのソースコードを Device Farm に提供する必要はありません。Android および iOS のアプリケーションは変更せずに送信できます。

セッション中のアプリケーションの再署名

Device Farm は、Android と iOS のアプリケーションの両方に再署名します。これにより、アプリケーションの署名に依存する機能が中断する可能性があります。例えば、Android の Google マップの API は、アプリケーションの署名に依存しています。アプリケーションの再署名を行うことで、Android デバイス用の DexGuard などの製品からの著作権侵害や不正使用を検出することもできます。

セッションでの難読化されたアプリケーション

Android アプリケーションでは、アプリケーションが難読化された場合、ProGuard を使用すれば、Device Farm でテストできます。ただし、DexGuard と著作権侵害対策を併用した場合、Device Farm では、アプリケーションの再署名を行えません。

AWS Device Farm でのプロジェクトによる作業

Device Farm の 1 つのプロジェクトは、複数の実行を含む、Device Farm 内の 1 つの論理ワークスペースのことで、それは、1 つ以上のデバイスに対する単一アプリケーションの各テストの 1 つの実行を表します。プロジェクトでは、お好きな形で複数のワークスペースを編成できます。例えば、1 つのプロジェクトは、アプリケーションタイトルに応じて、またはプラットフォームに応じて作れます。プロジェクトは必要な数だけ作成できます。

プロジェクトの取り扱いは、AWS Device Farm コンソール、AWS Command Line Interface (AWS CLI)、または AWS Device Farm API で行えます。

トピック

AWS Device Farm でプロジェクトを作成する

プロジェクトは AWS Device Farm コンソール、AWS CLI、または AWS Device Farm API を使用して作成できます。

前提条件

- 設定 のステップを完了します。

プロジェクトを作成する (コンソール)

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. [新規プロジェクト] を選択します。
4. プロジェクトの名前を入力し、[送信] を選択します。
5. プロジェクトの設定を指定するには、[プロジェクト設定] を選択します。これらの設定には、テスト実行のデフォルトのタイムアウトが含まれます。適用された設定は、プロジェクトのすべてのテスト実行で使用されます。詳細については、「AWS Device Farm でのテスト実行の実行タイムアウトを設定する」を参照してください。

プロジェクトを作成する (AWS CLI)

- プロジェクト名を指定して create-project を実行します。

  例:

  ```
  aws devicefarm create-project --name MyProjectName
  ```

  AWS CLI 応答には、プロジェクトの Amazon リソースネーム (ARN) が含まれます。

  ```
  {
      "project": {
          "name": "MyProjectName",
          "arn": "arn:aws:devicefarm:us-west-2:123456789101:project:5e01a8c7-
  c861-4c0a-b1d5-12345EXAMPLE",
          "created": 1535675814.414
      }
  }
  ```

  詳細については、create-project および AWS CLI リファレンス を参照してください。

プロジェクトを作成する (API)

- `CreateProject` API を呼び出します。

Device Farm API の使用についての詳細は、「Device Farm の自動化」を参照してください。

AWS Device Farm でプロジェクトリストを表示する

プロジェクトリストを表示するには、AWS Device Farm コンソール、AWS CLI、または AWS Device Farm API を使用します。

トピック

前提条件

- Device Farm で少なくとも 1 つのプロジェクトを作成します。「AWS Device Farm でプロジェクトを作成する」の指示に従ってから、このページに戻ります。

プロジェクトリストを表示する (コンソール)

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. 使用可能なプロジェクトのリストを見つけるには、以下を実行します:
   - モバイルデバイステストプロジェクトの場合、Device Farm ナビゲーションメニューで、[モバイルデバイスのテスト] を選択して、その後 [プロジェクト] を選択します。
   - デスクトップブラウザテストプロジェクトの場合、Device Farm ナビゲーションメニューには、[デスクトップブラウザのテスト] を選択して、その後 [プロジェクト] を選択します。

プロジェクトリストを表示する (AWS CLI)

- プロジェクトリストを表示するには、list-projects コマンドを実行します。

  1 つのプロジェクトについての情報を表示するには、get-project コマンドを実行します。

AWS CLI での Device Farm の使用についての詳細は、「AWS CLI リファレンス」を参照してください。

プロジェクトリストを表示する (API)

- プロジェクトリストを表示するには、`ListProjects` API を呼び出します。

  1 つのプロジェクトについての情報を表示するには、`GetProject` API を呼び出します。

AWS Device Farm API についての詳細は、「Device Farm の自動化」を参照してください。

AWS Device Farm でのテスト実行による作業

Device Farm の実行により、特定の一連のテストを使用して、特定の一連のデバイスで実行されるアプリケーションの特定のビルドが表されます。実行では、実行の結果に関する情報を含むレポートが作成されます。実行には、1 つ以上のジョブが含まれます。詳細については、「実行」を参照してください。

AWS Device Farm コンソール、 AWS Command Line Interface （ AWS CLI ）、または AWS Device Farm API を使用して、実行を操作できます。

トピック

Device Farm でのテスト実行の作成

Device Farm コンソール AWS CLI、または Device Farm API を使用してテスト実行を作成できます。また、サポートされているプラグイン (例: Device Farm 用の Jenkins または Gradle プラグイン) も使用できます。プラグインの詳細については、「ツールとプラグイン」を参照してください。実行に関しては、「実行」を参照してください。

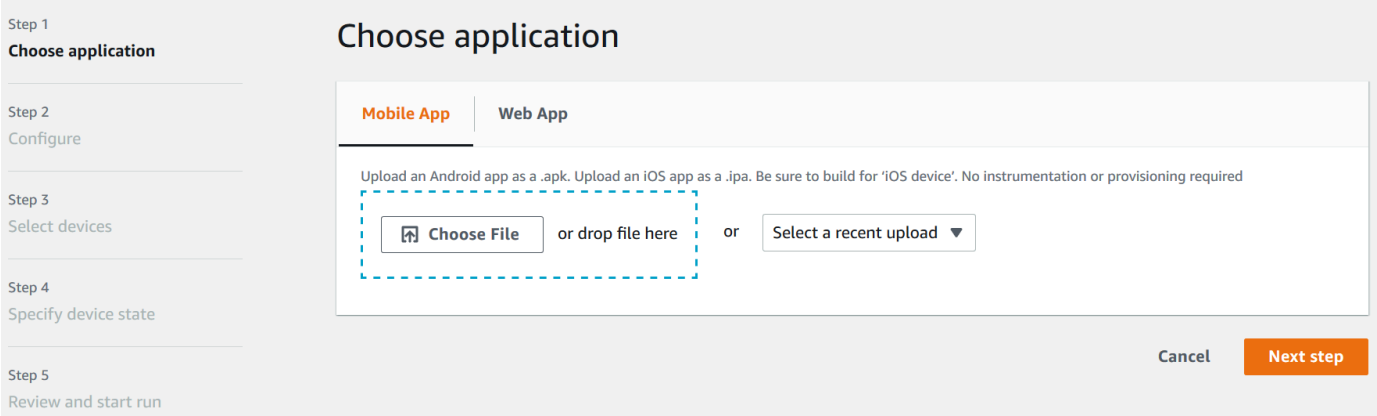
トピック

前提条件

Device Farm にプロジェクトが作成されている必要があります。「AWS Device Farm でプロジェクトを作成する」の指示に従ってから、このページに戻ります。

テスト実行を作成 (コンソール)

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. ナビゲーションペインで、 [モバイルデバイスのテスト] を選択し、そして次に、[プロジェクト] を選択します。
3. すでにプロジェクトがある場合は、テストをそのプロジェクトにアップロードできます。それ以外の場合は、[新規プロジェクト] を選択し、[プロジェクト名] を入力して、そして次に、[作成] を選択します。
4. プロジェクトを開き、[新規実行を作成 を選択します。
5. 「アプリケーションを選択する」ページで、[モバイルアプリケーション]、または [ウェブアプリケーション] を選択します。



6. アプリケーションファイルをアップロードします。ファイルをドラッグアンドドロップするか、最近のアップロードを選択することもできます。iOS アプリをアップロードする場合は、シミュレーターではなく、必ず [iOS デバイス] を選択してください。
7. (オプション) [実行名] に名前を入力します。デフォルトで、Device Farm はアプリケーションのファイル名を使用します。
8. [次へ] を選択します。
9. 「構成する」ページで、利用可能なテストスイートのいずれかを選択します。

> **Note**
>
> 利用可能なテストがない場合は、[ビルトイン: ファズ] を選択して、標準のビルトインテストスイートを実行します。[ビルトイン: ファズ] を選択して、[イベント数]、[イベント調整]、および [ランダマイザーシード] ボックスが表示されたら、値を変更するか、そのままにすることができます。

使用できるテストスイートの詳細については、「AWS Device Farm のテストタイプによる作業」を参照してください。

10. [ビルトイン: ファズ] を選択しなかった場合は、[ファイルを選択] を選んで、テストが含まれるファイルを参照して選択します。
11. テスト環境では、[標準環境でテストを実行] または [カスタム環境でテストを実行] を選択します。詳細については、「テスト環境」を参照してください。
12. 標準のテスト環境を使用している場合は、ステップ 13 にスキップします。デフォルトのテスト仕様 YAML ファイルでカスタムのテスト環境を使用している場合は、ステップ 13 にスキップします。

    a. カスタムのテスト環境でデフォルトのテスト仕様を編集する場合は、[編集] を選択して、デフォルトの YAML 仕様を更新します。

    b. テスト仕様を変更した場合、[新規として保存] を選択し、それを更新します。
13. 録画またはパフォーマンスデータキャプチャオプションを構成する場合は、[高度な構成] を選択します。

    a. テスト中に動画を記録にするには、[録画を有効化] を選択します。

    b. デバイスのパフォーマンスデータをキャプチャするには、[アプリケーションのパフォーマンスデータキャプチャを有効化] を選びます。

> **Note**
>
> プライベートデバイスがある場合は、[プライベートデバイス専用の構成] も表示されます。

14. [次へ を選択します。
15. 「デバイスを選択する」ページで、次のいずれかを実行します:

- ビルトインのデバイスプールを選択して [デバイスプール] に対してテストを実行するには、[上位デバイス] を選択します。
- 独自のデバイスプールを作成してそのテストを実行するには、「デバイスプールを作成」の手順に従って操作を行った後、このページに戻ります。
- [デバイスプール] ですでに独自のデバイスプールを作成した場合は、自分のデバイスプールを選択します。

詳細については、「AWS Device Farm でのデバイスサポート」を参照してください。

16. [次へ] を選択します。
17. 「デバイス状態を指定する」ページでは:
    - 実行中に Device Farm が使用する他のデータを提供するには、[別途データを追加] の横の [ファイルを選択 を選択後、データが含まれる .zip ファイルを参照して選択します。
    - 実行中に Device Farm が使用する追加のアプリケーションをインストールするには、[他のアプリをインストール] の横の [ファイルを選択] を選択し、アプリを含む .apk または .ipa ファイルを参照して選択します。インストールする他のアプリケーションにもこの手順を繰り返します。インストール順序を変更するには、アップロードした後にアプリケーションをドラッグアンドドロップします。
    - 実行中に Wi-Fi、Bluetooth、GPS、または NFC を有効にするかどうか指定するには、[無線状態を設定] の横にある適切なボックスを選択します。
    - 実行用のデバイスの緯度と経度をプリセットするには、[デバイスの場所] の横に座標を入力します。
    - 実行用のデバイスロケールをプリセットするには、[デバイスロケール] でロケールを選択します。
18. [次へ を選択します。
19. 「実行を確認して開始する」ページで、テスト実行の実行タイムアウトを指定できます。無制限のテストスロットを使用している場合は、[計測されていないスロットで実行] が選択されていることを確認します。
20. 実行タイムアウトを変更するには、値を入力するか、スライダーを使用します。詳細については、「AWS Device Farm でのテスト実行の実行タイムアウトを設定する」を参照してください。
21. [実行を確認して開始] を選択します。

Device Farm は、デバイスが利用可能になると、通常数分以内に実行を開始します。テスト実行中、Device Farm コンソールが実行テーブルに保留中アイコン

を表示します。実行中の各デバイスも保留中アイコンで起動し、テストが開始されると実行中アイコン

に切り替わります。各テストが終了すると、テスト結果アイコンがデバイス名の横に表示されます。すべてのテストが完了すると、実行の横にある保留中アイコンがテスト結果アイコンに変わります。

テスト実行を停止したい場合は、「AWS Device Farm での実行を停止する」を参照してください。

テスト実行を作成 (AWS CLI)

を使用してテスト実行 AWS CLI を作成できます。

トピック

- ステップ 1: プロジェクトを選択する
- ステップ 2: デバイスプールを選択する
- ステップ 3: アプリケーションファイルをアップロードする
- ステップ 4: テストスクリプトパッケージをアップロードする
- ステップ 5: (オプション) カスタムテスト仕様をアップロードする
- ステップ 6: テスト実行をスケジュール設定する

ステップ 1: プロジェクトを選択する

テスト実行は Device Farm プロジェクトに関連付ける必要があります。

1. Device Farm プロジェクトを一覧表示するには、list-projects を実行します。プロジェクトがない場合は、「AWS Device Farm でプロジェクトを作成する」を参照してください。

   例：

   ```
   aws devicefarm list-projects
   ```

   このレスポンスには、Device Farm プロジェクトのリストが含まれています。

   ```
   {
   ```

```
    "projects": [
        {
            "name": "MyProject",
            "arn": "arn:aws:devicefarm:us-west-2:123456789101:project:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
            "created": 1503612890.057
        }
    ]
}
```

2. プロジェクトを選択してテスト実行を関連付け、その Amazon Reソース Name (ARN) を書き留めます。

ステップ 2: デバイスプールを選択する

デバイスプールを選択して、テスト実行を関連付ける必要があります。

1. デバイスプールを表示するには、プロジェクト ARN を指定して list-device-pools を実行します。

   例：

   ```
   aws devicefarm list-device-pools --arn arn:MyProjectARN
   ```

   このレスポンスには、ビルトイン Device Farm デバイスプール (例: Top Devices や、このプロジェクト用に以前作成したデバイスプール) が含まれます。

   ```
   {
       "devicePools": [
           {
               "rules": [
                   {
                       "attribute": "ARN",
                       "operator": "IN",
                       "value": "[\"arn:aws:devicefarm:us-west-2::device:example1\",
   \"arn:aws:devicefarm:us-west-2::device:example2\",\"arn:aws:devicefarm:us-
   west-2::device:example3\"]"
                   }
               ],
               "type": "CURATED",
               "name": "Top Devices",
   ```

```
            "arn": "arn:aws:devicefarm:us-west-2::devicepool:example",
            "description": "Top devices"
        },
        {
            "rules": [
                {
                    "attribute": "PLATFORM",
                    "operator": "EQUALS",
                    "value": "\"ANDROID\""
                }
            ],
            "type": "PRIVATE",
            "name": "MyAndroidDevices",
            "arn": "arn:aws:devicefarm:us-west-2:605403973111:devicepool:example2"
        }
    ]
}
```

2. デバイスプールを選択し、その ARN を書き留めておきます。

   また、デバイスプールを作成し、その後、このステップに戻ることもできます。詳細については、「デバイスプールを作成する (AWS CLI)」を参照してください。

ステップ 3: アプリケーションファイルをアップロードする

アップロードリクエストを作成し、Amazon Simple Storage Service (Amazon S3) の署名付きアップロード URL を取得するには、以下が必要です:

- プロジェクト ARN。
- アプリケーションファイルの名前。
- アップロードのタイプ。

詳細については、「create-upload」を参照してください。

1. ファイルをアップロードするには、`--project-arn`、`--name`、および `--type` パラメータにより、create-upload を実行します。

   この例では、Android アプリケーション用のアップロードを作成します:

```
aws devicefarm create-upload --project-arn arn:MyProjectArn --name MyAndroid.apk --
type ANDROID_APP
```

このレスポンスには、アプリケーションアップロード ARN と署名付き URL が含まれます。

```
{
    "upload": {
        "status": "INITIALIZED",
        "name": "MyAndroid.apk",
        "created": 1535732625.964,
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "ANDROID_APP",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE"
    }
}
```

2. アプリケーションアップロード ARN と署名付き URL を書き留めます。
3. Amazon S3 の署名付き URL を使用してアプリケーションファイルをアップロードします。この例では、curl を使用して、Android .apk ファイルをアップロードします:

```
curl -T MyAndroid.apk "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL"
```

詳細については、「Amazon Simple Storage Service ユーザーガイド」の「署名付き URL を使用したオブジェクトのアップロード」を参照してください。

4. アプリケーションアップロードのステータスを確認するには、get-upload を実行し、アプリアップロードの ARN を指定します。

```
aws devicefarm get-upload --arn arn:MyAppUploadARN
```

レスポンスのステータスが SUCCEEDED になるまで待ってから、テストスクリプトパッケージをアップロードします。

```
{
    "upload": {
        "status": "SUCCEEDED",
```

```
        "name": "MyAndroid.apk",
        "created": 1535732625.964,
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "ANDROID_APP",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
        "metadata": "{"valid": true}"
    }
}
```

ステップ 4: テストスクリプトパッケージをアップロードする

次に、テストスクリプトパッケージをアップロードします。

1. アップロードリクエストを作成し、Amazon S3 の署名付きアップロード URL を取得するには、`--project-arn`、`--name`、および `--type` パラメータを使用して create-upload を実行します。

   この例では、Appium Java TestNG テストパッケージアップロードを作成します:

   ```
   aws devicefarm create-upload --project-arn arn:MyProjectARN --name MyTests.zip --
   type APPIUM_JAVA_TESTNG_TEST_PACKAGE
   ```

   このレスポンスには、テストパッケージのアップロード ARN および署名付き URL が含まれます。

   ```
   {
       "upload": {
           "status": "INITIALIZED",
           "name": "MyTests.zip",
           "created": 1535738627.195,
           "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
   ExampleURL",
           "type": "APPIUM_JAVA_TESTNG_TEST_PACKAGE",
           "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
   c861-4c0a-b1d5-12345EXAMPLE"
       }
   }
   ```

2. テストパッケージアップロード ARN と署名付き URL を書き留めます。

3. Amazon S3 の署名付き URL を使用して、テストスクリプトパッケージファイルをアップロードします。この例では、curl を使用して、圧縮された TestNG スクリプトファイルをアップロードします:

```
curl -T MyTests.zip "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL"
```

4. テストスクリプトパッケージアップロードのステータスを確認するには、get-upload を実行し、ステップ 1 からテストパッケージアップロード ARN を指定します。

```
aws devicefarm get-upload --arn arn:MyTestsUploadARN
```

レスポンスのステータスが SUCCEEDED となるまで待ってから、次のオプションのステップに進みます。

```
{
    "upload": {
        "status": "SUCCEEDED",
        "name": "MyTests.zip",
        "created": 1535738627.195,
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "APPIUM_JAVA_TESTNG_TEST_PACKAGE",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
        "metadata": "{"valid": true}"
    }
}
```

ステップ 5: (オプション) カスタムテスト仕様をアップロードする

標準のテスト環境でテストを実行している場合、このステップはスキップします。

Device Farm では、サポートされているテストタイプごとに、デフォルトのテスト仕様ファイルを保持しています。次に、デフォルトのテスト仕様をダウンロードし、カスタムのテスト環境でテストを実行するために必要なカスタムテスト仕様アップロードを作成します。詳細については、「テスト環境」を参照してください。

1. デフォルトのテスト仕様のアップロード ARN を検索するには、list-uploads を実行してプロジェクト ARN を指定します。

```
aws devicefarm list-uploads --arn arn:MyProjectARN
```

このレスポンスには、デフォルトのテスト仕様ごとのエントリが含まれます:

```
{
    "uploads": [
        {

            {
                "status": "SUCCEEDED",
                "name": "Default TestSpec for Android Appium Java TestNG",
                "created": 1529498177.474,
                "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
                "type": "APPIUM_JAVA_TESTNG_TEST_SPEC",
                "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE"
            }
        }
    ]
}
```

2. リストからデフォルトのテスト仕様を選択します。アップロード ARN を書き留めます。
3. デフォルトのテスト仕様をダウンロードするには、アップロード ARN を指定して get-upload を実行します。

   例：

   ```
   aws devicefarm get-upload --arn arn:MyDefaultTestSpecARN
   ```

   このレスポンスには、デフォルトのテスト仕様をダウンロードできる署名付き URL が含まれます。
4. この例では、curl を使用してデフォルトのテスト仕様をダウンロードし、MyTestSpec.yml という名前で保存します:

   ```
   curl "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/ExampleURL" >
    MyTestSpec.yml
   ```

5. デフォルトのテスト仕様を編集して、テスト要件を満たします。以降のテスト実行では変更後のテスト仕様を使用します。このステップをスキップして、デフォルトのテスト仕様をカスタムのテスト環境でそのまま使用します。
6. カスタムのテスト仕様のアップロードを作成するには、テスト仕様名、テスト仕様タイプ、およびプロジェクト ARN を指定して、create-upload を実行します。

   この例では、Appium Java TestNG のカスタムテスト仕様のアップロードを作成します:

   ```
   aws devicefarm create-upload --name MyTestSpec.yml --type
    APPIUM_JAVA_TESTNG_TEST_SPEC --project-arn arn:MyProjectARN
   ```

   このレスポンスには、テスト仕様のアップロード ARN および署名付き URL が含まれます:

   ```
   {
       "upload": {
           "status": "INITIALIZED",
           "category": "PRIVATE",
           "name": "MyTestSpec.yml",
           "created": 1535751101.221,
           "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
   ExampleURL",
           "type": "APPIUM_JAVA_TESTNG_TEST_SPEC",
           "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
   c861-4c0a-b1d5-12345EXAMPLE"
       }
   }
   ```

7. テスト仕様のアップロード ARN と署名付き URL を書き留めます。
8. Amazon S3 の署名付き URL を使用してテスト仕様ファイルをアップロードします。この例では、 curlを使用して Appium JavaTestNG テスト仕様をアップロードします。

   ```
   curl -T MyTestSpec.yml "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
   ExampleURL"
   ```

9. テスト仕様アップロードのステータスを確認するには、get-upload を実行し、アップロードの ARN を指定します。

   ```
   aws devicefarm get-upload --arn arn:MyTestSpecUploadARN
   ```

レスポンスのステータスが SUCCEEDED になるまで待ってから、テスト実行のスケジュール設定を行います。

```
{
    "upload": {
        "status": "SUCCEEDED",
        "name": "MyTestSpec.yml",
        "created": 1535732625.964,
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "APPIUM_JAVA_TESTNG_TEST_SPEC",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
        "metadata": "{"valid": true}"
    }
}
```

カスタムのテスト仕様をアップデートするには、このテスト仕様のアップロード ARN を指定して、update-upload を実行します。詳細については、「update-upload」を参照してください。

ステップ 6: テスト実行をスケジュール設定する

を使用してテスト実行をスケジュールするには AWS CLI、以下を指定して schedule-runを実行します。

- ステップ 1 からのプロジェクト ARN。
- ステップ 2 からのデバイスプール ARN。
- ステップ 3 からのアプリケーションアップロード ARN。
- ステップ 4 からのテストパッケージアップロード ARN。

カスタムテスト環境でテストを実行している場合は、ステップ 5 からのテスト仕様 ARN も必要です。

標準のテスト環境での実行をスケジュール設定するには

- プロジェクト ARN、デバイスプール ARN、アプリケーションアップロード ARN、テストパッケージ情報を指定して、schedule-run を実行します。

例：

```
aws devicefarm schedule-run --project-arn arn:MyProjectARN --app-
arn arn:MyAppUploadARN --device-pool-arn arn:MyDevicePoolARN --name MyTestRun --
test type=APPIUM_JAVA_TESTNG,testPackageArn=arn:MyTestPackageARN
```

このレスポンスには、テスト実行のステータスの確認に使用する実行 ARN が含まれます。

```
{
    "run": {
        "status": "SCHEDULING",
        "appUpload": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345appEXAMPLE",
        "name": "MyTestRun",
        "radios": {
            "gps": true,
            "wifi": true,
            "nfc": true,
            "bluetooth": true
        },
        "created": 1535756712.946,
        "totalJobs": 179,
        "completedJobs": 0,
        "platform": "ANDROID_APP",
        "result": "PENDING",
        "devicePoolArn": "arn:aws:devicefarm:us-
west-2:123456789101:devicepool:5e01a8c7-c861-4c0a-b1d5-12345devicepoolEXAMPLE",
        "jobTimeoutMinutes": 150,
        "billingMethod": "METERED",
        "type": "APPIUM_JAVA_TESTNG",
        "testSpecArn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345specEXAMPLE",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:run:5e01a8c7-c861-4c0a-
b1d5-12345runEXAMPLE",
        "counters": {
            "skipped": 0,
            "warned": 0,
            "failed": 0,
            "stopped": 0,
            "passed": 0,
            "errored": 0,
            "total": 0
```

```
        }
    }
}
```

詳細については、「schedule-run」を参照してください。

カスタムテスト環境での実行をスケジュール設定するには

- ステップは、--test パラメータの追加 testSpecArn 属性を伴い、標準テスト環境用のステップとほとんど同じです。

  例：

```
aws devicefarm schedule-run --project-arn arn:MyProjectARN --app-
arn arn:MyAppUploadARN --device-pool-arn arn:MyDevicePoolARN --name MyTestRun --
test
 testSpecArn=arn:MyTestSpecUploadARN,type=APPIUM_JAVA_TESTNG,testPackageArn=arn:MyTestPacka
```

テスト実行のステータスを確認するには

- get-run コマンドを使い、以下の実行 ARN を指定します:

```
aws devicefarm get-run --arn arn:aws:devicefarm:us-
west-2:111122223333:run:5e01a8c7-c861-4c0a-b1d5-12345runEXAMPLE
```

詳細については、「get-run」を参照してください。で Device Farm を使用する方法については、AWS CLI「」を参照してくださいAWS CLI リファレンス。

テスト実行を作成 (API)

手順は、 AWS CLI セクションで説明されている手順と同じです。テスト実行を作成 (AWS CLI) を参照してください。

以下の情報は、ScheduleRun API を呼び出すために必要です:

- プロジェクト ARN。「プロジェクトを作成する (API)」および「CreateProject」を参照してください。
- アプリケーションアップロード ARN。CreateUpload を参照してください。

- テストパッケージアップロード ARN。CreateUpload を参照してください。
- デバイスプール ARN。「デバイスプールを作成」および「CreateDevicePool」を参照してください。

> **Note**
>
> カスタムのテスト環境でテストを実行している場合は、テスト仕様アップロード ARN も必要です。詳細については、「ステップ 5: (オプション) カスタムテスト仕様をアップロードする」および「CreateUpload」を参照してください。

Device Farm APIの使用についての詳細は、「Device Farm の自動化」を参照してください。

次のステップ

Device Farm コンソールでテスト実行が完了すると、クロックアイコン

が成功などの結果アイコン

に変わります。テストが完了した時点で実行のレポートが表示されます。詳細については、「AWS Device Farm でのレポート」を参照してください。

レポートを使用するには、「Device Farm でのテストレポートによる作業」の手順に従います。

AWS Device Farm でのテスト実行の実行タイムアウトを設定する

各デバイスのテストの実行を停止するまでのテスト実行の実行時間の値を設定することができます。デフォルトの実行タイムアウトはデバイスあたり 150 分ですが、5 分などの低い値にも設定できます。AWS Device Farm コンソール AWS CLI、または AWS Device Farm API を使用して、実行タイムアウトを設定できます。

> **Important**
>
> 実行タイムアウトオプションは、一部のバッファと共に、テスト実行の最大所要時間に設定する必要があります。例えば、テストにデバイスあたり 20 分かかる場合、デバイスあたり 30 分のタイムアウトを選択する必要があります。

実行がタイムアウトを超えた場合、そのデバイスでの実行は強制的に停止されます。可能であれば、部分的な結果を使用することができます。従量制課金オプションを使用している場合は、その時点までの実行に対して請求されます。料金の詳細については、「Device Farm 料金表」を参照してください。

各デバイスでテスト実行にかかる時間を知っている場合は、この機能を使用できます。テスト実行の実行タイムアウトを指定すると、何らかの理由でテスト実行が滞り、実行されているテストがないデバイス分に対して課金されるという状況を回避できます。つまり、実行タイムアウト機能を使用すると、テスト実行に予想以上の時間がかかっている場合にその実行を停止できます。

実行タイムアウトは、プロジェクトレベルおよびテスト実行レベルの 2 つの場所で設定できます。

前提条件

1. 「設定」のステップを完了します。
2. Device Farm でプロジェクトを作成します。「AWS Device Farm でプロジェクトを作成する」の指示に従ってから、このページに戻ります。

プロジェクトの実行タイムアウトを設定する

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. 既にプロジェクトがある場合は、リストからそのプロジェクトを選択します。それ以外の場合は、[新規プロジェクト] を選択し、プロジェクト名を入力して、[送信] を選択します。
4. [プロジェクト設定] を選択します。
5. [全般] タブの [実行タイムアウト] に値を入力するか、スライダーバーを使用します。
6. [保存] を選択します。

   プロジェクト内のテスト実行で、先ほど指定した実行タイムアウト値が使用されるようになりました。ただし、実行をスケジュール設定する際にタイムアウト値を上書きする場合は除きます。

テスト実行の実行タイムアウトを設定する

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。

2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. 既にプロジェクトがある場合は、リストからそのプロジェクトを選択します。それ以外の場合は、[新規プロジェクト] を選択し、プロジェクト名を入力して、[送信] を選択します。
4. [新規実行を作成] を選択します。
5. ステップに従って、アプリケーションを選択し、テストを構成し、デバイスを選択し、デバイス状態を指定します。
6. [実行を確認して開始] の [実行タイムアウトを設定] では、値を入力するか、スライダーバーを使用します。
7. [実行を確認して開始] を選択します。

AWS Device Farm 実行用のネットワークの接続と条件をシミュレートする

Device Farm で Android、iOS、FireOS、およびウェブアプリケーションのテストをしながら、ネットワークシェーピングを使用して、ネットワークの接続と条件をシミュレートできます。例えば、ネットワーク条件が完全でなくてもアプリケーションをテストすることができます。

デフォルトのネットワーク設定を使用して実行を作成した場合は、各デバイスは、完全で制限のない WiFi 接続でインターネット接続できます。ネットワークシェーピングを使用すると、Wi-Fi 接続を変更して、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の両方のスループット、遅延、ジッター、損失を制御する 3G や Lossy WiFi などのネットワークプロファイルを指定できます。

トピック

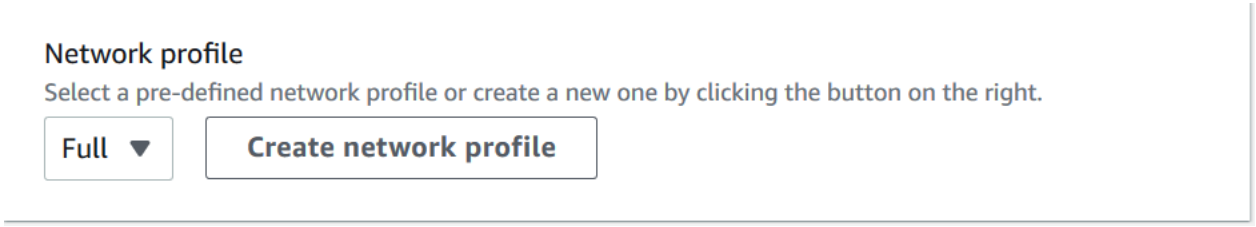
- テスト実行をスケジュールする場合のネットワークシェーピングを設定する
- ネットワークプロファイルを作成する
- テスト中にネットワーク条件を変更する

テスト実行をスケジュールする場合のネットワークシェーピングを設定する

実行をスケジュールする場合、Device Farm でキュレートされたプロファイルから選択するか、独自のプロファイルを作成および管理できます。

1. 任意の Device Farm プロジェクトから、[新規実行を作成] を選択します。

   まだプロジェクトがない場合は、「AWS Device Farm でプロジェクトを作成する」を参照してください。
2. アプリケーションを選択後、[次へ] を選択します。
3. テストを構成し、[次へ] を選択します。
4. デバイスを選択し、[次へ] を選択します。
5. [ロケーションとネットワークを設定] セクションで、ネットワークプロファイルを選択するか、[ネットワークプロファイルを作成] を選択して、独自のものを作成します。

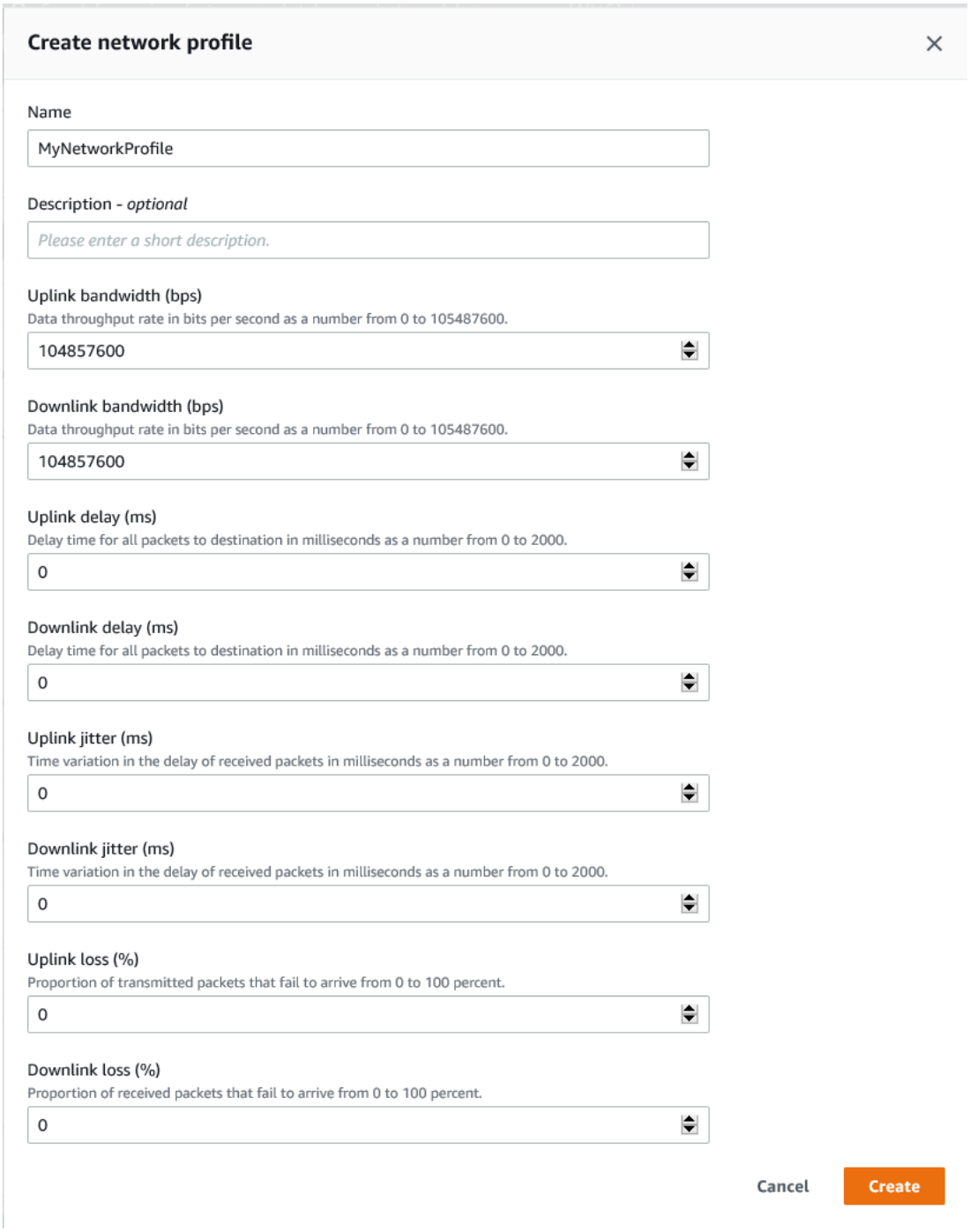


6. [次へ] を選択します。
7. テスト実行を確認して開始します。

ネットワークプロファイルを作成する

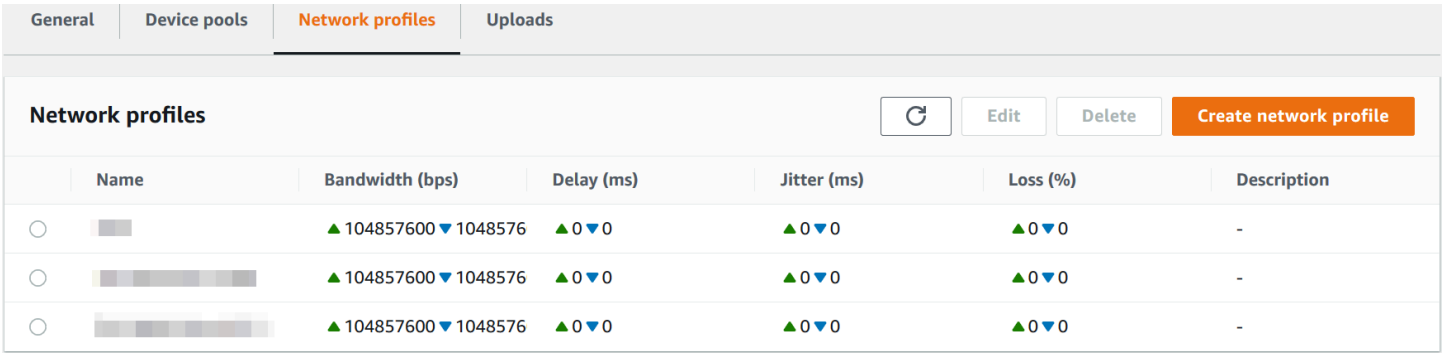
テスト実行の作成時に、ネットワークプロファイルを作成できます。

1. [ネットワークプロファイルを作成] を選択します。



2. ネットワークプロファイルの名前と設定を入力します。
3. [作成] を選択します。
4. テスト実行の作成を完了し、実行を開始します。

ネットワークプロファイルを作成したら、「プロジェクト設定」ページで表示および管理できるようになります。



テスト中にネットワーク条件を変更する

テスト実行中の帯域幅の減少といった動的ネットワーク条件をシミュレートするには、Appium などのフレームワークを使用してデバイスホストから API を呼び出します。詳細については、「」を参照してくださいCreateNetworkProfile。

AWS Device Farm での実行を停止する

実行の開始後に停止が必要となることがあります。例えば、テストの実行中に問題が発生し、更新されたテストスクリプトにより実行を再開する場合です。

Device Farm コンソール AWS CLI、または API を使用して実行を停止できます。

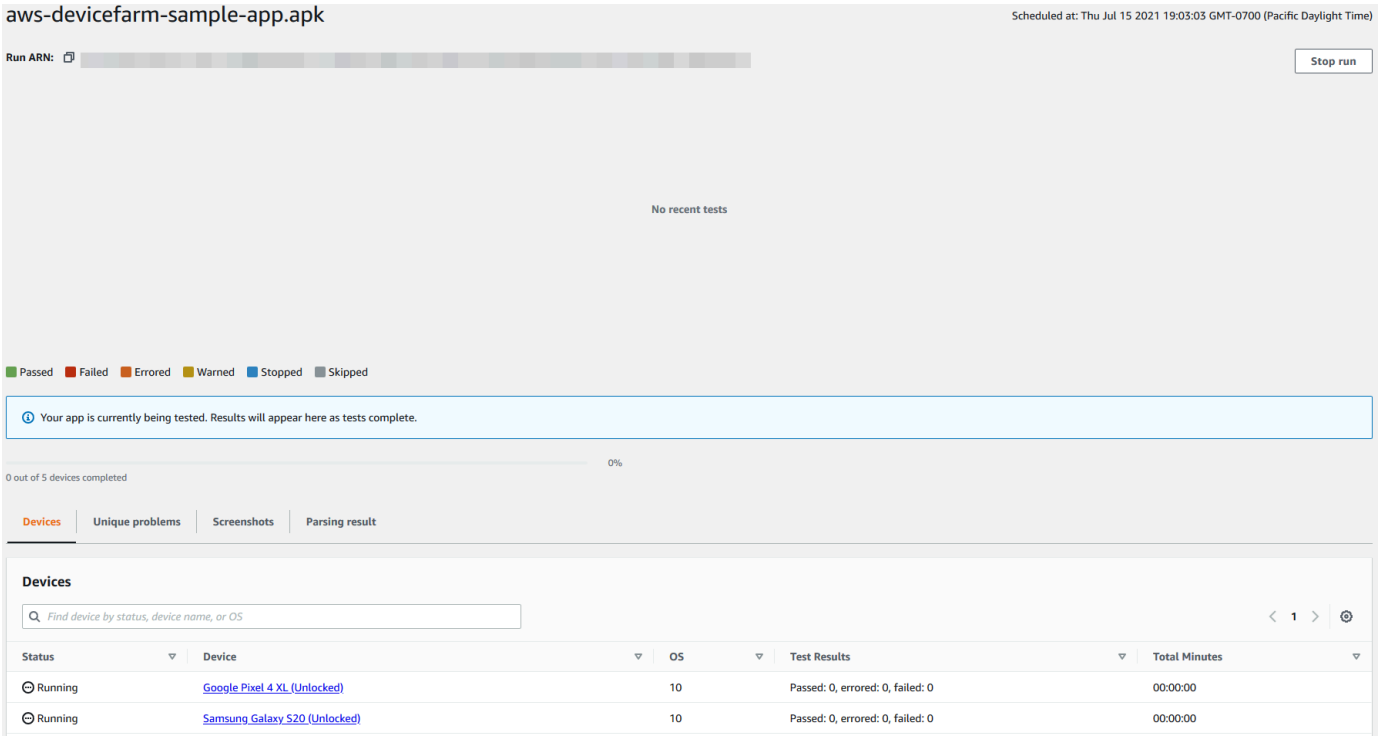
トピック

- 実行を停止する (コンソール)
- 実行を停止 (AWS CLI)
- 実行を停止 (API)

実行を停止する (コンソール)

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. アクティブなテスト実行を持つプロジェクトを選択します。
4. 「自動テスト」ページで、テスト実行を選択します。

   デバイス名の左に、保留中または実行中の アイコンが表示されます。

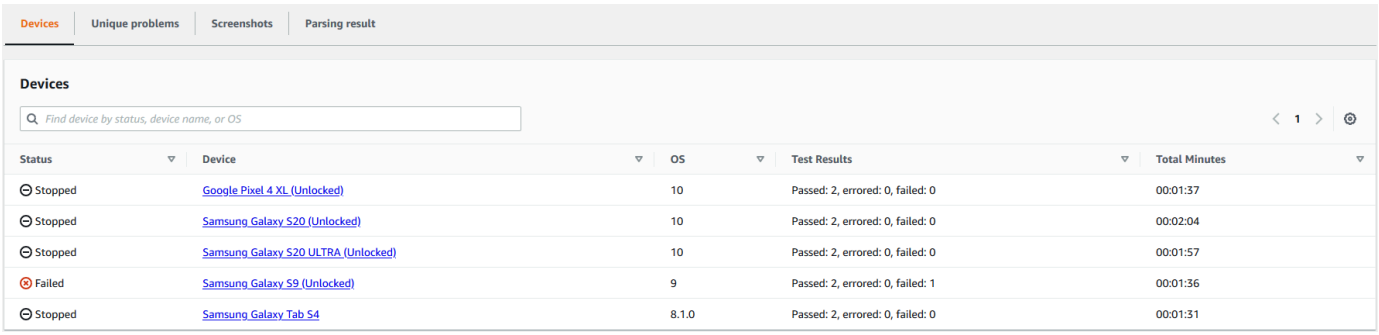


5. [実行を停止] を選択します。

しばらくすると、マイナスのマークを赤色の円で囲んだアイコンがデバイス名の横に表示されます。実行が停止すると、アイコンの色が赤から黒に変わります。

> **Important**
>
> テストは一旦実行されると、Device Farm により停止できません。進行中の場合、Device Farm はテストを停止します。請求対象の合計時間が [デバイス] セクションに表示されます。さらに請求は、Device Farm がセットアップスイートとティアダウンスイートの実行に費やす合計時間に対しても行われます。詳細については、「Device Farm 料金表」を参照してください。

次の図は、テストが正常に停止した後に表示される [デバイス] セクションの例です。



実行を停止 (AWS CLI)

次のコマンドを実行して、指定されたテスト実行を停止することもできます。*myARN* はテスト実行の Amazon リソースネーム (ARN) です。

```
$ aws devicefarm stop-run --arn myARN
```

次のような出力が表示されます:

```
{
    "run": {
        "status": "STOPPING",
        "name": "Name of your run",
        "created": 1458329687.951,
        "totalJobs": 7,
        "completedJobs": 5,
        "deviceMinutes": {
            "unmetered": 0.0,
            "total": 0.0,
            "metered": 0.0
        },
        "platform": "ANDROID_APP",
        "result": "PENDING",
        "billingMethod": "METERED",
        "type": "BUILTIN_EXPLORER",
        "arn": "myARN",
        "counters": {
            "skipped": 0,
            "warned": 0,
            "failed": 0,
            "stopped": 0,
            "passed": 0,
```

```
            "errored": 0,
            "total": 0
        }
    }
}
```

実行の ARN を取得するには、`list-runs` コマンドを使用します。出力は次の例のようになります:

```
{
    "runs": [
        {
            "status": "RUNNING",
            "name": "Name of your run",
            "created": 1458329687.951,
            "totalJobs": 7,
            "completedJobs": 5,
            "deviceMinutes": {
                "unmetered": 0.0,
                "total": 0.0,
                "metered": 0.0
            },
            "platform": "ANDROID_APP",
            "result": "PENDING",
            "billingMethod": "METERED",
            "type": "BUILTIN_EXPLORER",
            "arn": "Your ARN will be here",
            "counters": {
                "skipped": 0,
                "warned": 0,
                "failed": 0,
                "stopped": 0,
                "passed": 0,
                "errored": 0,
                "total": 0
            }
        }
    ]
}
```

で Device Farm を使用する方法については、 AWS CLI「」を参照してくださいAWS CLI リファレンス。

実行を停止 (API)

- テスト実行に StopRunオペレーションを呼び出します。

Device Farm API の使用についての詳細は、「Device Farm の自動化」を参照してください。

AWS Device Farm で実行のリストを表示する

Device Farm コンソール、または API を使用して AWS CLI、プロジェクトの実行のリストを表示できます。

トピック

実行のリストを表示する (コンソール)

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. プロジェクトのリストで、表示するリストに対応するプロジェクトを選択します。

> **Tip**
>
> 検索バーを使って、名前によりプロジェクトリストを絞り込めます。

実行のリストを表示する (AWS CLI)

- list-runs コマンドを実行します。

  単一の実行についての情報を表示するには、get-run コマンドを実行します。

で Device Farm を使用する方法については、 AWS CLI「」を参照してくださいAWS CLI リファレンス。

実行のリストを表示する (API)

- `ListRuns` API を呼び出します。

  単一の実行についての情報を表示するには、`GetRun` API を呼び出します。

Device Farm API についての詳細は、「Device Farm の自動化」を参照してください。

AWS Device Farm でデバイスプールを作成する

Device Farm コンソール AWS CLI、または API を使用して、デバイスプールを作成できます。

トピック

前提条件

- Device Farm コンソールで実行を作成します。「Device Farm でのテスト実行の作成」の手順に従います。「デバイスを選択する」ページが表示されたら、このセクションの手順に進みます。

デバイスプールを作成 (コンソール)

1. 「デバイスを選択する」ページで、[デバイスプールを作成する] を選択します。
2. [名前] に、このデバイスプールの分かりやすい名前を入力します。
3. [説明] に、このデバイスプールの分かりやすい説明を入力します。
4. このデバイスプール内のデバイスに対して 1 つ以上の選択条件を使用する場合は、次の手順を行います:

   a. [動的デバイスプールの作成] を選択します。

   b. [ルールを追加] を選択します。

   c. [フィールド] (最初のドロップダウンリスト) で、 次のいずれかを選択します:

- デバイスをメーカー名ごとに含めるには、[デバイスメーカー] を選択します。
- デバイスをタイプの値ごとに含めるには、[フォームファクター] を選択します。

d. [演算子] (2 番目のドロップダウンリスト) で [EQUALS] を選択して、[フィールド] と [値] の値が等しいデバイスを含めます。

e. [値] (3 番目のドロップダウンリスト) では、[フィールド] 値と [演算子] 値に指定する値を入力または選択します。[フィールド] で [プラットフォーム] を選ぶ場合、指定できるのは [ANDROID] と [IOS] のみです。同様に、[フィールド] で [フォームファクター] を選ぶ場合、指定できるセクションは [電話] と [タブレット] のみです。

f. 別のルールを追加するには、[ルールを追加] を選択します。

g. ルールを削除するには、削除するルールの横にある [X] アイコンを選択します。

最初のルールを作成すると、デバイスのリストで、ルールに一致する各デバイスの横にあるボックスが選択されます。ルールを作成または変更すると、デバイスのリストで、それらの結合されたルールに一致する各デバイスの横にあるボックスが選択されます。ボックスが選択されているデバイスはデバイスプールに含まれます。ボックスが選択解除されたデバイスは除外されます。

5. 個々のデバイスを手動で含めたり除外したりする場合は、以下を実行します:

a. [静的デバイスプールを作成] を選択します。

b. 各デバイスの横にあるボックスの選択または選択解除をします。ルールを指定していない場合にのみ、ボックスの選択または選択解除をできます。

6. 表示されているすべてのデバイスを含めたり除外したりする場合は、リストの列ヘッダー行のボックスの選択または選択解除をします。

> **Important**
>
> 列ヘッダー行のボックスを使用して表示されたデバイスのリストを変更することはできますが、残りの表示されたデバイスのみが含まれたり除外されたりするわけではありません。含まれる、または除外されるデバイスを確認するには、列ヘッダー行のすべてのボックスのコンテンツを選択解除してから、ボックスを参照します。

7. [作成] を選択します。

デバイスプールを作成する (AWS CLI)

- create-device-pool コマンドを実行します。

で Device Farm を使用する方法については、 AWS CLI「」を参照してくださいAWS CLI リファレンス。

デバイスプールを作成する (API)

- `CreateDevicePool` API を呼び出します。

Device Farm API の使用についての情報は、「Device Farm の自動化」を参照してください。

AWS Device Farm での結果の分析

標準テスト環境では、Device Farm コンソールを使用して、テスト実行の各テストのレポートを表示することができます。

また、Device Farm は、テスト実行完了時にダウンロードできるファイル、ログ、画像といった他のアーティファクトも収集します。

トピック

- Device Farm でのテストレポートによる作業
- Device Farm でのアーティファクトによる作業

Device Farm でのテストレポートによる作業

テストレポートを表示するには、Device Farm コンソールを使用します。詳細については、「AWS Device Farm でのレポート」を参照してください。

トピック

- 前提条件
- テスト結果の把握
- レポートの表示

前提条件

テスト実行を設定し、完了したことを確認します。

1. 実行を作成するには「Device Farm でのテスト実行の作成」を参照し、その後、このページに戻ります。
2. 実行が完了したことを確認します。テスト実行中、Device Farm コンソールでは進行中のものに保留中アイコン

   が表示されます。実行中の各デバイスも保留中のアイコンで
   起動し、テストが開始されると実行中の

   アイコンに切り替わります。各テストが終了すると、テスト結果アイコンがデバイス名の横に表示されます。すべてのテストが完了すると、実行の横にある保留中アイコンがテスト結果アイコンに変わります。詳細については、「テスト結果の把握」を参照してください。

テスト結果の把握

Device Farm コンソールが表示するアイコンは、完了したテスト実行の状態の迅速な評価に役立ちます。

トピック

- 個々のテストの結果のレポート
- 複数テストの結果のレポート

個々のテストの結果のレポート

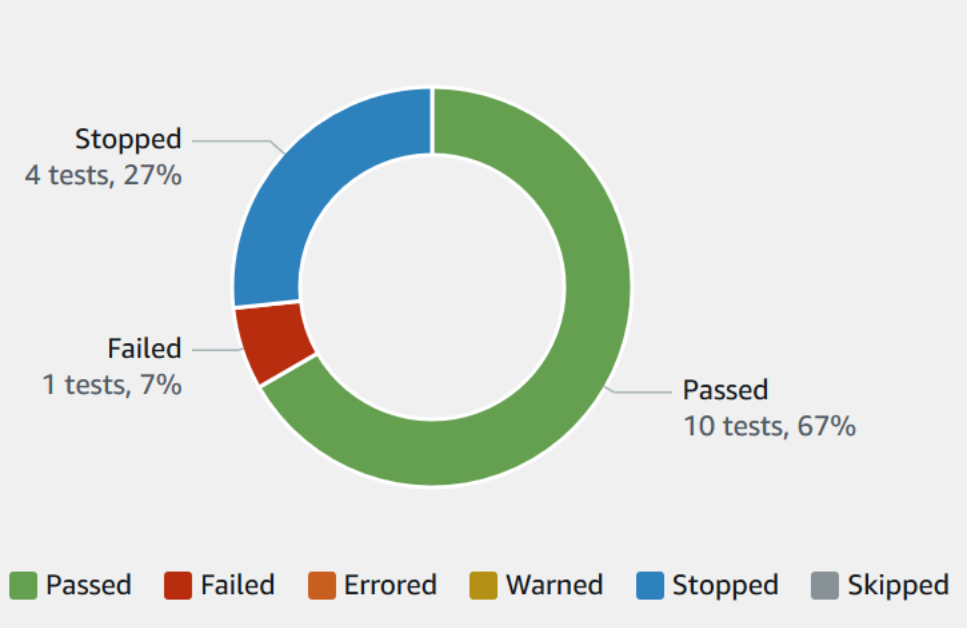
個々のテストについて伝えるレポートで、Device Farm は以下のアイコンを表示します:

説明	アイコン
テストが成功しました。	
テストが失敗しました。	
Device Farm がテストをスキップしました。	

説明	アイコン
テストが停止しました。	
Device Farm が警告が返しました。	
Device Farm がエラーを返しました。	

複数テストの結果のレポート

完了した実行を選択すると、Device Farm がテスト結果の概要グラフを表示します。



例えば、このテスト実行の結果グラフは、停止したテストが 4、失敗したテストが 1、成功したテストが 10 あることを表します。

グラフには常に色分けとラベル付けがされています。

レポートの表示

テストの結果は、Device Farm コンソールで表示できます。

トピック

- テスト実行の概要ページの表示

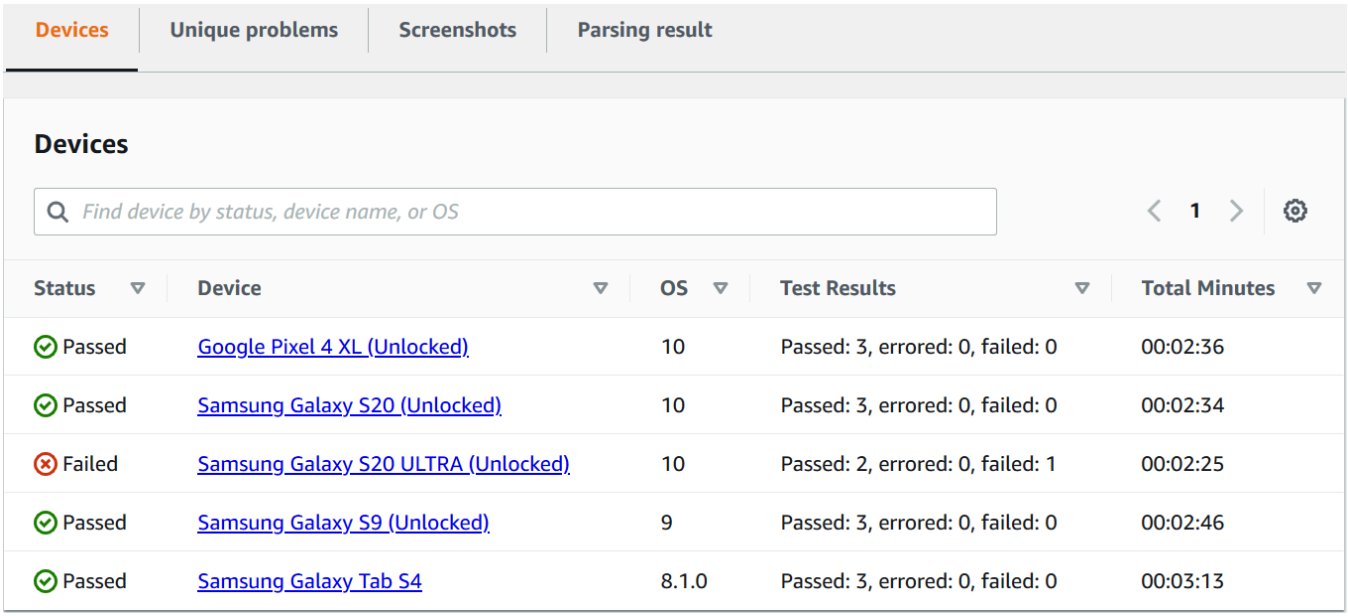
- 一意の問題のレポートを表示する
- デバイスのレポートを表示する
- テストスイートレポートを表示する
- テストレポートを表示する
- レポート内の問題、デバイス、スイート、またはテストのパフォーマンスデータを表示する
- レポート内の問題、デバイス、スイート、またはテストのログ情報を表示する

テスト実行の概要ページの表示

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. ナビゲーションペインで、 [モバイルデバイスのテスト] を選択し、次に、[プロジェクト] を選択します。
3. プロジェクトのリストで、実行のプロジェクトを選択します。

> **Tip**
> 名前によりプロジェクトリストを絞り込むには、検索バーを使用します。

4. 概要レポートページを表示するには、完了した実行を選択します。
5. テスト実行の概要ページに、テスト結果の概要が表示されます。
   - 「一意の問題] セクションには、固有の警告と障害がリストされています。一意の問題を表示するには、「一意の問題のレポートを表示する」の手順に従います。
   - 「デバイス] セクションには、各デバイスでの合計テスト数が結果ごとに表示されます。



この例には、いくつかのデバイスがあります。最初のテーブルエントリで、Android バージョン 10 を実行する Google Pixel 4 XL デバイスのレポートによると、3 件の成功したテストは実行に 2 分 36 秒を要しました。

デバイスごとに結果を表示するには、「デバイスのレポートを表示する」の手順に従います。

- 「スクリーンショット] セクションには、実行中に Device Farm がキャプチャーしてデバイスごとにグループ化した、スクリーンショットのリストが表示されます。
- 「解析結果] セクションでは、解析結果をダウンロードできます。

一意の問題のレポートを表示する

1. [一意の問題] で、表示する問題を選択します。
2. デバイスを選択します。レポートには、問題に関する情報が表示されます。

   「動画] セクションには、ダウンロード可能なテストのテストの動画記録が表示されます。

   [結果] セクションには、テストの結果が表示されます。ステータスは結果アイコンとして表されます。詳細については、「個々のテストの結果のレポート」を参照してください。

   [ログ] セクションには、テスト中に Device Farm が記録した情報が表示されます。この情報を表示するには、「レポート内の問題、デバイス、スイート、またはテストのログ情報を表示する」の手順に従います。

[パフォーマンス] タブには、テスト中に Device Farm が生成したパフォーマンスデータに関する情報が表示されます。このパフォーマンスデータを表示するには、「レポート内の問題、デバイス、スイート、またはテストのパフォーマンスデータを表示する」の手順に従います。

[ファイル] タブには、ダウンロード可能な、テスト関連ファイル (ログファイルなど) の一覧が表示されます。ファイルをダウンロードするには、リスト内のファイルのリンクを選択します。

[スクリーンショット] タブには、Device Farm がテスト中にキャプチャした、スクリーンショットのリストが表示されます。

デバイスのレポートを表示する

- [デバイス] セクションで、デバイスを選択します。

  [ビデオ] セクションには、ダウンロード可能なテストのビデオ記録が表示されます。

  [スイート] セクションには、デバイスのスイートに関する情報を含んでいるテーブルが表示されます。

  このテーブルでは、[テスト結果] 列に、デバイス上で実行した各テストスイートにおけるテスト数を結果ごとにまとめています。また、このデータにはグラフィカルコンポーネントもあります。詳細については、「複数テストの結果のレポート」を参照してください。

  スイートごとにすべての結果を表示するには、「テストスイートレポートを表示する」の手順に従います。

  [ログ] セクションには、実行中に Device Farm がデバイス用に記録した情報が表示されます。この情報を表示するには、「レポート内の問題、デバイス、スイート、またはテストのログ情報を表示する」の手順に従います。

  [パフォーマンス] セクションには、実行中に Device Farm がデバイス用に生成したパフォーマンスデータに関する情報が表示されます。このパフォーマンスデータを表示するには、「レポート内の問題、デバイス、スイート、またはテストのパフォーマンスデータを表示する」の手順に従います。

  [ファイル] セクションには、デバイスの他、ダウンロード可能な関連ファイル (ログファイルなど) に関するスイートのリストが表示されます。ファイルをダウンロードするには、リスト内のファイルのリンクを選択します。

[スクリーンショット] セクションには、デバイスの実行中に Device Farm がキャプチャしてスイートごとにグループ化した、スクリーンショットのリストが表示されます。

テストスイートレポートを表示する

1. [デバイス] セクションで、デバイスを選択します。
2. [スイート] セクションで、テーブルからスイートを選択します。

   [ビデオ] セクションには、ダウンロード可能なテストのビデオ記録が表示されます。

   [テスト] セクションには、スイートのテストに関する情報を含んでいるテーブルが表示されます。

   テーブルでは、[テスト結果] 列に結果が表示されます。また、このデータにはグラフィカルコンポーネントもあります。詳細については、「複数テストの結果のレポート」を参照してください。

   テストごとに結果を最大限に表示するには、「テストレポートを表示する」の手順に従います。

   [ログ] セクションには、スイートの実行中に Device Farm が記録した情報が表示されます。この情報を表示するには、「レポート内の問題、デバイス、スイート、またはテストのログ情報を表示する」の手順に従います。

   [パフォーマンス] セクションには、スイートの実行中に Device Farm が生成したパフォーマンスデータに関する情報が表示されます。このパフォーマンスデータを表示するには、「レポート内の問題、デバイス、スイート、またはテストのパフォーマンスデータを表示する」の手順に従います。

   [ファイル] セクションには、スイート用のテストのリストと、ダウンロード可能な関連ファイル (ログファイルなど) が表示されます。ファイルをダウンロードするには、リスト内のファイルのリンクを選択します。

   [スクリーンショット] セクションには、スイートの実行中に Device Farm がキャプチャしてテストごとにグループ化した、スクリーンショットのリストが表示されます。

テストレポートを表示する

1. [デバイス] セクションで、デバイスを選択します。

2. [スイート] セクションで、スイートを選択します。
3. [テスト] セクションで、テストを選択します。
4. [ビデオ] セクションには、ダウンロード可能なテストのビデオ記録が表示されます。

   [結果] セクションには、テストの結果が表示されます。ステータスは結果アイコンとして表されます。詳細については、「個々のテストの結果のレポート」を参照してください。

   [ログ] セクションには、テスト中に Device Farm が記録した情報が表示されます。この情報を表示するには、「レポート内の問題、デバイス、スイート、またはテストのログ情報を表示する」の手順に従います。

   [パフォーマンス] タブには、テスト中に Device Farm が生成したパフォーマンスデータに関する情報が表示されます。このパフォーマンスデータを表示するには、「レポート内の問題、デバイス、スイート、またはテストのパフォーマンスデータを表示する」の手順に従います。

   [ファイル] タブには、ダウンロード可能な、テスト関連ファイル (ログファイルなど) の一覧が表示されます。ファイルをダウンロードするには、リスト内のファイルのリンクを選択します。

   [スクリーンショット] タブには、Device Farm がテスト中にキャプチャした、スクリーンショットのリストが表示されます。

レポート内の問題、デバイス、スイート、またはテストのパフォーマンスデータを表示する

> **Note**
>
> Device Farm は、現時点では Android デバイスのみ デバイスのパフォーマンスデータを収集します。

以下の情報が [パフォーマンス] タブに表示されます:

- [CPU] グラフは、選択された問題、デバイス、スイート、またはテストにおいてアプリケーションが 1 つのコアで使用した CPU の割合 (縦軸) を時間 (横軸) に渡って表示します。

  縦軸は、0% から最大記録値までパーセンテージで表します。

  アプリケーションが複数のコアを使用する場合、パーセンテージは 100% を超えることがあります。例えば、3 つのコアが 60% 使用されている場合は 180% と表示されます。

- [メモリ] グラフは、グラフは、選択された問題、デバイス、スイート、またはテストにおいてアプリケーションが使用した MB 数 (縦軸) を時間 (横軸) に渡って表示します。

  縦軸は、0 MB から最大記録値まで MB で表します。
- [スレッド] グラフは、選択された問題、デバイス、スイート、またはテストにおいて使用したスレッド数 (縦軸) を時間 (横軸) に渡って表示します。

  縦軸は、0 スレッドから最大記録値までスレッド数で表します。

いずれの場合も横軸は、選択された問題、デバイス、スイート、またはテストの実行の開始から終了までを秒単位で表します。

特定のデータポイントの情報を表示するには、対象のグラフで横軸に沿いながら該当する秒で一時停止します。

レポート内の問題、デバイス、スイート、またはテストのログ情報を表示する

[ログ] セクションでは以下の情報が表示されます:

- ソース] はログエントリのソースを表します。可能な値は以下のとおりです:
  - [ハーネス] は、Device Farm が作成したログエントリを表します。これらのログエントリは、通常、イベントの開始および停止中に作成されます。
  - [デバイス] は、デバイスが作成したログエントリを表します。Android の場合、これらのログエントリは logcat と互換性があります。iOS の場合、これらのログエントリは syslog と互換性があります。
  - テスト] は、テストまたはテストフレームワークのいずれかが作成したログエントリを表します。
- 時間] は、最初のログエントリと、このログエントリの間の経過時間を表します。時間は *MM:SS.SSS* 形式で表されます。*M* は分を表し、*S* は秒を表します。
- PID] は、ログエントリを作成したプロセス識別子 (PID) を表します。デバイス上のアプリケーションによって作成されるログエントリの PID はすべて同一になります。
- レベル] はログエントリのロギングレベルを表します。例えば、`Logger.debug("This is a message!")` は Debug のレベルを記録します。使用できる値は次のとおりです:
  - アラート
  - [非常事態]
  - デバッグ

- 緊急
- エラー
- エラー発生
- 失敗
- 情報
- 内部
- 注意
- 成功
- スキップ
- 停止
- 詳細
- 警告済み
- 警告

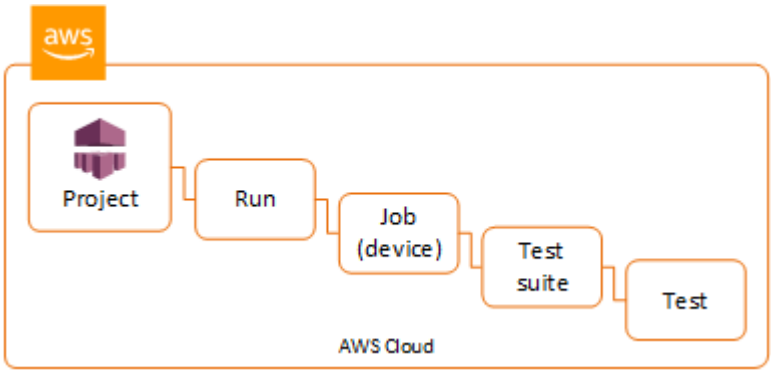
- タグ] は、ログエントリの任意のメタデータを表します。例えば、Android logcat は、システムのどの部分がログエントリを作成したかを記述するためにこれを使用できます (例: `ActivityManager`)。
- メッセージ] は、ログエントリのメッセージまたはデータを表します。例えば、`Logger.debug("Hello, World!")` は `"Hello, World!"` のメッセージを記録します。

情報の一部のみを表示するには:

- 特定の列の値と一致するすべてのログエントリを表示するには、検索バーに値を入力します。例えば、Harness の [ソース] 値を使用してすべてのログエントリを表示するには、検索バーに「**Harness**」と入力します。
- 列ヘッダーボックスからすべての文字を削除するには、その列ヘッダーボックスの [X] を選択します。列ヘッダーボックスからすべての文字を削除することは、その列のヘッダーボックスに「*****」を入力するのと同じです。

実行したすべてのスイートとテストを含む、デバイスのすべてのログ情報をダウンロードするには、[ログをダウンロード] を選択します。

Device Farm でのアーティファクトによる作業



Device Farm では、レポート、ログファイル、画像などのアーティファクトを各実行テストで収集します。

テスト実行中に作成されたアーティファクトはダウンロードできます:

ファイル

テスト実行中に生成されたファイル (例: Device Farm レポート)。詳細については、「Device Farm でのテストレポートによる作業」を参照してください。

ログ

テスト実行の各テストの出力。

スクリーンショット

テスト実行のテストごとに記録されるスクリーン画像。

アーティファクトの使用 (コンソール)

1. テスト実行のレポートページで、[デバイス] からモバイルデバイスを選択します。
2. ファイルをダウンロードするには、[ファイル] からいずれかを選択します。
3. テスト実行からログをダウンロードするには、[ログ] から [ログをダウンロード] を選択します。
4. スクリーンショットをダウンロードするには、[スクリーンショット] からスクリーンショットを選択します。

カスタムのテスト環境におけるアーティファクトのダウンロードの詳細については、「カスタムテスト環境でのアーティファクトの使用」を参照してください。

アーティファクトの使用 (AWS CLI)

を使用して AWS CLI 、テスト実行アーティファクトを一覧表示できます。

トピック

ステップ 1: Amazon リソースネーム (ARN) を取得する

アーティファクトは、実行、ジョブ、テストスイート、またはテストごとにリストできます。対応する ARN を指定する必要があります。この表は、各 AWS CLI リストコマンドの入力 ARN を示しています。

AWS CLI コマンドを一覧表示する	必須 ARN
list-projects	このコマンドは、すべてのプロジェクトを返し、ARN を必要としません。
list-runs	`project`
list-jobs	`run`
list-suites	`job`
list-tests	`suite`

例えば、テスト ARN を見つけるには、テストスイート ARN を入力パラメータとして使用して list-tests を実行します。

例：

```
aws devicefarm list-tests --arn arn:MyTestSuiteARN
```

この応答には、テストスイートにある各テストのテスト ARN が含まれます。

```
{
```

```
    "tests": [
        {
            "status": "COMPLETED",
            "name": "Tests.FixturesTest.testExample",
            "created": 1537563725.116,
            "deviceMinutes": {
                "unmetered": 0.0,
                "total": 1.89,
                "metered": 1.89
            },
            "result": "PASSED",
            "message": "testExample passed",
            "arn": "arn:aws:devicefarm:us-west-2:123456789101:test:5e01a8c7-c861-4c0a-
b1d5-12345EXAMPLE",
            "counters": {
                "skipped": 0,
                "warned": 0,
                "failed": 0,
                "stopped": 0,
                "passed": 1,
                "errored": 0,
                "total": 1
            }
        }
    ]
}
```

ステップ 2: アーティファクトをリストする

AWS CLI list-artifacts コマンドは、ファイル、スクリーンショット、ログなどのアーティファクトのリストを返します。各アーティファクトには URL が含まれ、ファイルをダウンロードできます。

- 実行、ジョブ、テストスイート、またはテスト ARN を指定して、list-artifacts を呼び出します。タイプ (ファイル、ログ、またはスクリーンショット) を指定します。

  この例は、各テストで使用できる各アーティファクトの ダウンロード用 URL を返します:

  ```
  aws devicefarm list-artifacts --arn arn:MyTestARN --type "FILE"
  ```

  この応答には、各アーティファクトのダウンロード用 URL が含まれます。

  ```
  {
  ```

```
    "artifacts": [
        {
            "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
            "extension": "txt",
            "type": "APPIUM_JAVA_OUTPUT",
            "name": "Appium Java Output",
            "arn": "arn:aws:devicefarm:us-west-2:123456789101:artifact:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
        }
    ]
}
```

ステップ 3: アーティファクトをダウンロードする

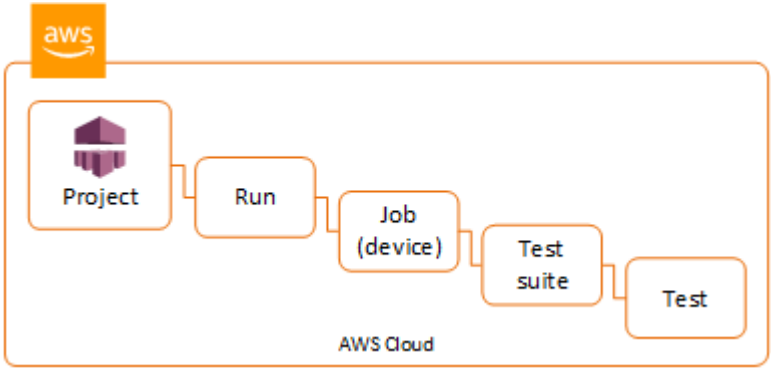
- 前のステップの URL を使用して、アーティファクトをダウンロードします。この例では、curl を使用して、Android Appium Java 出力ファイルをダウンロードします:

```
curl "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/ExampleURL"
 > MyArtifactName.txt
```

アーティファクトの使用 (API)

Device Farm API ListArtifacts メソッドは、ファイル、スクリーンショット、ログなどのアーティファクトのリストを返します。各アーティファクトには URL が含まれ、ファイルをダウンロードできます。

カスタムテスト環境でのアーティファクトの使用



カスタムテスト環境で、Device Farm は、カスタムレポート、ログファイル、画像などのアーティファクトを収集します。これらのアーティファクトは、テスト実行でデバイスごとに表示されます。

テスト実行中に作成されるこれらのアーティファクトはダウンロードできます:

テスト仕様出力

テスト仕様 YAML ファイル内のコマンドの実行による出力。

お客様のアーティファクト

テスト実行のアーティファクトを含む ZIP ファイル。テスト仕様 YAML ファイルの [アーティファクト:] セクションで構成されます。

テスト仕様シェルスクリプト

YAML ファイルから作成される中間シェルスクリプト。このシェルスクリプトファイルはテスト実行で使用されるため、YAML ファイルのデバッグに使用できます。

テスト仕様ファイル

テスト実行で使用される YAML ファイル。

詳細については、「Device Farm でのアーティファクトによる作業」を参照してください。

AWS Device Farm リソースのタギング

AWS Device Farm は AWS Resource Groups Tagging API と連携します。この API を使用すると、タグで AWS アカウントのリソースを管理できます。プロジェクトやテスト実行などのリソースにタグを追加できます。

タグを使用して以下のことができます:

- 独自のコスト構造を反映するように AWS 請求情報を整理します。そのためには、AWS アカウントにサインアップして、タグキー値が含まれた AWS アカウント請求書を取得します。次に、結合したリソースのコストを見るには、同じタグキー値のリソースに従って請求書情報を整理します。例えば、複数のリソースにアプリケーション名のタグを付け、請求情報を整理することで、複数のサービスに渡るそのアプリケーションの合計コストを確認できます。詳細については、「AWS の請求情報とコスト管理」の「コスト配分とタギング」を参照してください。
- IAM ポリシーを通じてアクセスをコントロールします。そのためには、タグ値条件を使用してリソースまたはリソースのセットへのアクセスを許可するポリシーを作成します。
- タグとして特定プロパティ (テストに使用されたブランチなど) が設定された実行を識別および管理します。

リソースのタギングの詳細については、「タギングベストプラクティス」ホワイトペーパーを参照してください。

トピック

リソースのタギング

AWS Resource Group Tagging API を使用すると、リソースのタグを追加、削除、または変更できます。詳細については、「AWS Resource Group Tagging API リファレンス」を参照してください。

リソースにタグ付けするには、`resourcegroupstaggingapi` エンドポイントから `TagResources` オペレーションを使用します。このオペレーションでは、サポートされるサービスから ARN のリストとキー値ペアのリストを取得します。値はオプションです。空の文字列は、その

タグに値がないことを示します。例えば、以下の Python のサンプルタグでは、一連のプロジェクト ARN にタグ build-config を付けて、値 release を指定しています:

```
import boto3

client = boto3.client('resourcegroupstaggingapi')

client.tag_resources(ResourceARNList=["arn:aws:devicefarm:us-
west-2:111122223333:project:123e4567-e89b-12d3-a456-426655440000",
                                      "arn:aws:devicefarm:us-
west-2:111122223333:project:123e4567-e89b-12d3-a456-426655441111",
                                      "arn:aws:devicefarm:us-
west-2:111122223333:project:123e4567-e89b-12d3-a456-426655442222"]
                     Tags={"build-config":"release", "git-commit":"8fe28cb"})
```

タグ値は必須ではありません。値のないタグを設定するには、値を指定するときに空の文字列 ("") を使用します。タグは 1 つの値のみ保持できます。タグがリソースに対して保持する以前の値は、新しい値で上書きされます。

タグによるリソースの検索

タグによりリソースを検索するには、resourcegrouptaggingapi エンドポイントから GetResources オペレーションを使用します。このオペレーションは一連のフィルターを受け取りますが、いずれも必須ではなく、指定された条件に一致するリソースを返します。フィルタがない場合、すべてのタグ付きリソースが返されます。GetResources オペレーションでは、以下の条件に基づいてリソースをフィルタリングできます。

- タグ値
- リソースタイプ (devicefarm:run など)

詳細については、「AWS Resource Group Tagging API リファレンス」を参照してください。

以下の例では、値が production であるタグ stack を使用して Device Farm デスクトップブラウザテストセッション (devicefarm:testgrid-session リソース) を検索します:

```
import boto3
client = boto3.client('resourcegroupstaggingapi')
sessions = client.get_resources(ResourceTypeFilters=['devicefarm:testgrid-session'],
                                TagFilters=[
```

```
                                    {"Key":"stack","Values":["production"]}
                                ])
```

リソースからのタグの削除

タグを削除するには、削除するリソースとタグのリストを指定し、UntagResources オペレーションを使用します:

```
import boto3
client = boto3.client('resourcegroupstaggingapi')
client.UntagResources(ResourceARNList=["arn:aws:devicefarm:us-
west-2:111122223333:project:123e4567-e89b-12d3-a456-426655440000"], TagKeys=["RunCI"])
```

AWS Device Farm のテストタイプによる作業

このセクションでは、テストフレームワークとビルトインのテストタイプに対する Device Farm のサポートについて説明します。

テストフレームワーク

Device Farm では、以下の自動化テストフレームワークをサポートしています:

Android アプリケーションテストフレームワーク

- Appium と AWS Device Farm による作業
- Android および AWS Device Farm のインストゥルメンテーションによる作業

iOS アプリケーションテストフレームワーク

- Appium と AWS Device Farm による作業
- iOS 用 XCTest と AWS Device Farm による作業
- XCTest UI

ウェブアプリケーションテストフレームワーク

ウェブアプリケーションは、Appium を使用してサポートされます。テストを Appium に持ち込む方法の詳細については、「Appium と AWS Device Farm による作業」を参照してください。

カスタムテスト環境のフレームワーク

Device Farm では、XCTest フレームワークのテスト環境のカスタマイズをサポートしていません。詳細については、「カスタムテスト環境による作業」を参照してください。

Appium バージョンのサポート

Device Farm では、カスタム環境で実行されるテスト向けに Appium バージョン 1 をサポートしています。詳細については、「テスト環境」を参照してください。

ビルトインテストタイプ

ビルトインテストでは、テスト自動化スクリプトを記述、管理することなく、複数デバイスでアプリケーションをテストできます。Device Farm は 1 つのビルトインテストタイプを提供します:

- ビルトイン: ファズ (Android および iOS)

Appium と AWS Device Farm による作業

このセクションでは、Appium テストを構成して、パッケージし、 Device Farm にアップロードする方法について説明します。Appium は、ネイティブおよびモバイル型のウェブアプリケーションを自動化するためのオープンソースのツールです。詳細については、Appium ウェブサイト上の「Appium の紹介」を参照してください。

サンプルアプリケーションと動作テストへのリンクについては、 の「Android 用 Device Farm サンプルアプリケーション」および「iOS 用 Device Farm サンプルアプリケーション」を参照してください GitHub。

バージョンのサポート

さまざまなフレームワークやプログラミング言語のサポートは、使用する言語によって異なります。

Device Farm では、Appium サーバーバージョン 1.x および 2.x をすべてサポートしています。Android では、`devicefarm-cli` を含む Appium の主要バージョンであればどれでも選択できます。例えば、Appium サーバーバージョン 2 を使用するには、テスト仕様の YAML ファイルにこれらのコマンドを追加します:

```
phases:
  install:
    commands:
      # To install a newer version of Appium such as version 2:
      - export APPIUM_VERSION=2
      - devicefarm-cli use appium $APPIUM_VERSION
```

iOS では、avm コマンドか npm コマンドを使用して特定の Appium バージョンを選択できます。例えば、avm コマンドを使用して Appium サーバーバージョンを 2.1.2 に設定するには、テスト仕様の YAML ファイルにこれらのコマンドを追加します:

```
phases:
```

```
  install:
    commands:
      # To install a newer version of Appium such as version 2.1.2:
      - export APPIUM_VERSION=2.1.2
      - avm $APPIUM_VERSION
```

npm コマンドを使って Appium 2 の最新バージョンを使用するには、テスト仕様の YAML ファイルにこれらのコマンドを追加します:

```
phases:
  install:
    commands:
      - export APPIUM_VERSION=2
      - npm install -g appium@$APPIUM_VERSION
```

`devicefarm-cli` か他の CLI コマンドの詳細については、「AWS CLI コマンドリファレンス」を参照してください。

注釈など、フレームワークのすべての機能を使用するには、カスタムテスト環境を選択し、AWS CLI または `Device Farm` コンソールを使用して、カスタムテスト仕様をアップロードします。

トピック

- Appium テストパッケージを構成する
- 圧縮テストパッケージファイルを作成する
- テストパッケージを Device Farm にアップロードする
- テストのスクリーンショットを撮る (オプション)

Appium テストパッケージを構成する

テストパッケージを構成するには、次の手順を実行します。

Java (JUnit)

1. `pom.xml` を変更して、パッケージを JAR ファイルに設定します:

```
<groupId>com.acme</groupId>
<artifactId>acme-myApp-appium</artifactId>
<version>1.0-SNAPSHOT</version>
```

```
<packaging>jar</packaging>
```

2. テストを JAR ファイルにビルドするよう、pom.xml を変更して maven-jar-plugin を使用します。

   次のプラグインは、テストソースコード (src/test ディレクトリ内のもの) を JAR ファイルにビルドします:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.6</version>
  <executions>
    <execution>
      <goals>
        <goal>test-jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

3. 依存関係を JAR ファイルとしてビルドするよう、pom.xml を変更して maven-dependency-plugin を使用します。

   次のプラグインは依存関係を dependency-jars ディレクトリにコピーします:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>2.10</version>
  <executions>
    <execution>
      <id>copy-dependencies</id>
      <phase>package</phase>
      <goals>
        <goal>copy-dependencies</goal>
      </goals>
      <configuration>
        <outputDirectory>${project.build.directory}/dependency-jars/</
outputDirectory>
      </configuration>
    </execution>
  </executions>
```

```
</plugin>
```

4. 次の XML アセンブリを src/main/assembly/zip.xml に保存します。

   次の XML は、構成時に、Maven がビルド出力ディレクトリと dependency-jars ディレクトリのルートにあるすべてを含む .zip ファイルをビルドするように指示するアセンブリ定義です:

```
<assembly
    xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/
assembly/1.1.0 http://maven.apache.org/xsd/assembly-1.1.0.xsd">
  <id>zip</id>
  <formats>
    <format>zip</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory>./</outputDirectory>
      <includes>
        <include>*.jar</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory>./</outputDirectory>
      <includes>
        <include>/dependency-jars/</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```

5. テストとすべての依存関係を単一の .zip ファイルにパッケージするよう、pom.xml を変更して maven-assembly-plugin を使用します。

   次のプラグインは、上記のアセンブリを使用して、mvn package が実行されるたびに、ビルド出力ディレクトリに zip-with-dependencies という名前の .zip ファイルを作成します:

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.5.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <finalName>zip-with-dependencies</finalName>
        <appendAssemblyId>false</appendAssemblyId>
        <descriptors>
          <descriptor>src/main/assembly/zip.xml</descriptor>
        </descriptors>
      </configuration>
    </execution>
  </executions>
</plugin>
```

> **Note**
>
> 注釈が 1.3 でサポートされていないというエラーが表示された場合は、以下を pom.xml に追加します:
>
> ```
> <plugin>
>   <artifactId>maven-compiler-plugin</artifactId>
>   <configuration>
>     <source>1.7</source>
>     <target>1.7</target>
>   </configuration>
> </plugin>
> ```

Java (TestNG)

1. pom.xml を変更して、パッケージを JAR ファイルに設定します

```
<groupId>com.acme</groupId>
```

```
<artifactId>acme-myApp-appium</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
```

2. テストを JAR ファイルにビルドするよう、pom.xml を変更して maven-jar-plugin を使用します。

   次のプラグインは、テストソースコード (src/test ディレクトリ内のもの) を JAR ファイルにビルドします:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.6</version>
  <executions>
    <execution>
      <goals>
        <goal>test-jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

3. 依存関係を JAR ファイルとしてビルドするよう、pom.xml を変更して maven-dependency-plugin を使用します。

   次のプラグインは依存関係を dependency-jars ディレクトリにコピーします:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>2.10</version>
  <executions>
    <execution>
      <id>copy-dependencies</id>
      <phase>package</phase>
      <goals>
        <goal>copy-dependencies</goal>
      </goals>
      <configuration>
        <outputDirectory>${project.build.directory}/dependency-jars/</
outputDirectory>
      </configuration>
```

```
    </execution>
  </executions>
</plugin>
```

4. 次の XML アセンブリを src/main/assembly/zip.xml に保存します。

   次の XML は、構成時に、Maven がビルド出力ディレクトリと dependency-jars ディレクトリのルートにあるすべてを含む .zip ファイルをビルドするように指示するアセンブリ定義です:

```
<assembly
    xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/
assembly/1.1.0 http://maven.apache.org/xsd/assembly-1.1.0.xsd">
  <id>zip</id>
  <formats>
    <format>zip</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory>./</outputDirectory>
      <includes>
        <include>*.jar</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory>./</outputDirectory>
      <includes>
        <include>/dependency-jars/</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```

5. テストとすべての依存関係を単一の .zip ファイルにパッケージするよう、pom.xml を変更して maven-assembly-plugin を使用します。

次のプラグインは、上記のアセンブリを使用して、mvn package が実行されるたびに、ビルド出力ディレクトリに zip-with-dependencies という名前の .zip ファイルを作成します:

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.5.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <finalName>zip-with-dependencies</finalName>
        <appendAssemblyId>false</appendAssemblyId>
        <descriptors>
          <descriptor>src/main/assembly/zip.xml</descriptor>
        </descriptors>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Note

注釈が 1.3 でサポートされていないというエラーが表示された場合は、以下を pom.xml に追加します:

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.7</source>
    <target>1.7</target>
  </configuration>
</plugin>
```

Node.JS

Appium Node.js テストをパッケージして Device Farm にアップロードするには、ローカルマシンに次のものをインストールする必要があります:

- Node Version Manager (nvm)

  不要な依存関係がテストパッケージに含まれないように、テストを開発およびパッケージするときにこのツールを使用してください。
- Node.js
- npm-bundle (グローバルにインストール済み)

1. nvm が存在することを確認します。

   ```
   command -v nvm
   ```

   出力として nvm が表示されるはずです。

   詳細については、「」の「nvm」を参照してください GitHub。
2. Node.js をインストールするには、このコマンドを実行します:

   ```
   nvm install node
   ```

   特定バージョンの Node.js を指定できます:

   ```
   nvm install 11.4.0
   ```
3. 正しいバージョンのノードが使用されていることを確認します:

   ```
   node -v
   ```
4. npm-bundle をグローバルにインストールします:

   ```
   npm install -g npm-bundle
   ```

Python

1. 不要な依存関係がアプリケーションパッケージに含まれないように、テストの開発とパッケージのために Python virtualenv を設定することを強くお勧めします。

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

> **Tip**
>
> - グローバルサイトパッケージディレクトリからパッケージを継承するため、`--system-site-packages` オプションを使用して Python virtualenv を作成しないでください。テストで不要な依存関係を仮想環境に含めることになる場合があります。
> - これらのネイティブライブラリは、これらのテストが実行されるインスタンス上に存在する場合と存在しない場合があるため、ネイティブライブラリに依存する依存関係をテストで使用しないことも確認する必要があります。

2. 仮想環境に py.test をインストールします。

```
$ pip install pytest
```

3. Appium Python クライアントを仮想環境にインストールします。

```
$ pip install Appium-Python-Client
```

4. カスタムモードで別のパスを指定しない限り、Device Farm はテストが `tests/` に格納されると想定します。`find` を使用して、フォルダ内のすべてのファイルを表示できます:

```
$ find tests/
```

これらのファイルに、Device Farm で実行するテストスイートが含まれていることを確認します。

```
tests/
tests/my-first-tests.py
tests/my-second-tests/py
```

5. 仮想環境のワークスペースフォルダからこのコマンドを実行して、テストを実行せずにテストのリストを表示します。

```
$ py.test --collect-only tests/
```

Device Farm で実行するテストが出力に表示されていることを確認します。

6. tests/ folder 下にあるすべてのキャッシュファイルを消去します:

```
$ find . -name '__pycache__' -type d -exec rm -r {} +
$ find . -name '*.pyc' -exec rm -f {} +
$ find . -name '*.pyo' -exec rm -f {} +
$ find . -name '*~' -exec rm -f {} +
```

7. ワークスペースで次のコマンドを実行して、requirements.txt ファイルを生成します:

```
$ pip freeze > requirements.txt
```

Ruby

Appium Ruby テストをパッケージして Device Farm にアップロードするには、ローカルマシンに次のものをインストールする必要があります:

- Ruby Version Manager (RVM)

  不要な依存関係がテストパッケージに含まれないように、テストを開発およびパッケージするときにこのコマンドラインツールを使用してください。
- Ruby
- Bundler (この Gem には通常、Ruby がすでにインストールされています)。

1. 必要なキー、RVM、および Ruby をインストールします。手順については、RVM ウェブサイトの「RVM のインストール」を参照してください。

   インストールが完了したら、サインアウトしてから再度サインインして端末を再リロードします。

> **Note**
>
> RVM は bash シェル専用の関数としてロードされます。

2. rvm が正しくインストールされたことを確認します。

```
command -v rvm
```

出力として `rvm` が表示されるはずです。

3. 特定バージョンの Ruby (例えば *`2.5.3`*) をインストールする場合は、次のコマンドを実行してください:

```
rvm install ruby 2.5.3 --autolibs=0
```

リクエストされた Ruby のバージョンを使用していることを確認します:

```
ruby -v
```

4. 対象のテストプラットフォーム用のパッケージをコンパイルするようにバンドラーを構成します:

```
bundle config specific_platform true
```

5. .lock ファイルを更新して、テストの実行に必要なプラットフォームを追加します。

   - Android デバイスで実行するようにテストをコンパイルする場合は、次のコマンドを実行して Gemfile が Android テストホストの依存関係を使用するように構成します:

     ```
     bundle lock --add-platform x86_64-linux
     ```

   - iOS デバイスで実行するようにテストをコンパイルする場合は、次のコマンドを実行して、iOS テストホストの依存関係を使用するように Gemfile を構成します:

     ```
     bundle lock --add-platform x86_64-darwin
     ```

6. 通常、bundler gem はデフォルトでインストールされます。そうでない場合は、インストールします:

```
gem install bundler -v 2.3.26
```

圧縮テストパッケージファイルを作成する

> **Warning**
>
> Device Farm では、圧縮されたテストパッケージ内のファイルのフォルダ構造が重要であり、一部のアーカイブツールでは ZIP ファイルの構造が暗黙的に変更されます。ローカルデスクトップのファイルマネージャー (Finder や Windows エクスプローラーなど) に組み込まれているアーカイブユーティリティを使用するよりも、以下に指定されているコマンドラインユーティリティに従うことをお勧めします。

次に、Device Farm のテストをバンドルします。

Java (JUnit)

テストのビルドとパッケージ:

```
$ mvn clean package -DskipTests=true
```

その結果 `zip-with-dependencies.zip` のファイルが作成されます。これはお客様のテストパッケージです。

Java (TestNG)

テストのビルドとパッケージ:

```
$ mvn clean package -DskipTests=true
```

その結果 `zip-with-dependencies.zip` のファイルが作成されます。これはお客様のテストパッケージです。

Node.JS

1. プロジェクトをチェックアウトします。

   プロジェクトのルートディレクトリにいることを確認します。 ルートディレクトリで `package.json` を確認できます。

2. ローカルの依存関係をインストールするには、このコマンドを実行します。

```
npm install
```

このコマンドは、現在のディレクトリ内に node_modules フォルダも作成します。

> **Note**
>
> この時点で、ローカルでテストを実行できるようにする必要があります。

3. このコマンドを実行して、現在のフォルダ内のファイルを *.tgz ファイルにパッケージします。ファイルの名前は、package.json ファイルの name プロパティを使用して付けられます。

```
npm-bundle
```

この tarball(.tgz) ファイルには、コードと依存関係がすべて含まれています。

4. このコマンドを実行して、前のステップで生成した tarball (*.tgz ファイル) を単一の zip アーカイブにバンドルします:

```
zip -r MyTests.zip *.tgz
```

これは、次の手順で Device Farm にアップロードする MyTests.zip ファイルです。

Python

Python 2

pip を使用して、必要な Python パッケージ (「ホイールハウス」と呼ばれる) のアーカイブを生成します:

```
$ pip wheel --wheel-dir wheelhouse -r requirements.txt
```

ホイールハウス、テスト、pip 要件を Device Farm の zip アーカイブにパッケージします:

```
$ zip -r test_bundle.zip tests/ wheelhouse/ requirements.txt
```

Python 3

テストと pip 要件を zip ファイルにパッケージします:

```
$ zip -r test_bundle.zip tests/ requirements.txt
```

Ruby

1. 仮想 Ruby 環境を作成するには、次のコマンドを実行します:

```
# myGemset is the name of your virtual Ruby environment
rvm gemset create myGemset
```

2. 先ほど作成した環境を使用するには、次のコマンドを実行します:

```
rvm gemset use myGemset
```

3. ソースコードを確認してください。

   プロジェクトのルートディレクトリにいることを確認します。 ルートディレクトリで Gemfile を確認できます。

4. ローカルの依存関係と、Gemfile からのすべての Gem をインストールするには、このコマンドを実行します:

```
bundle install
```

> **Note**
>
> この時点で、ローカルでテストを実行できるようにする必要があります。テストをローカルで実行するには、このコマンドを使用します:
>
> ```
> bundle exec $test_command
> ```

5. vendor/cache フォルダの Gem をパッケージします。

```
# This will copy all the .gem files needed to run your tests into the vendor/
cache directory
bundle package --all-platforms
```

6. 次のコマンドを実行して、すべての依存関係とともにソースコードを単一の zip アーカイブにバンドルします:

```
zip -r MyTests.zip Gemfile vendor/ $(any other source code directory files)
```

これは、次の手順で Device Farm にアップロードする `MyTests.zip` ファイルです。

テストパッケージを Device Farm にアップロードする

Device Farm コンソールを使用してテストをアップロードできます。

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. 新規ユーザーである場合は、[新規プロジェクト] を選択し、プロジェクト名を入力してから、[送信] を選択します。

   すでにプロジェクトがある場合は、それを選択して、テストをそのプロジェクトにアップロードできます。
4. プロジェクトを開き、[新規実行を作成] を選択します。
5. ネイティブの Android と iOS テストの場合

   「アプリケーションを選択する」ページで、[モバイルアプリケーション] を選択し、その後 [ファイルを選択] を選び、アプリケーションの配布可能パッケージをアップロードします。

   > **Note**
   >
   > ファイルは Android `.apk` または iOS `.ipa` のいずれかである必要があります。iOS アプリケーションは、シミュレーターではなく、実際のデバイス用に構築される必要があります。

   モバイルウェブアプリケーションのテストの場合

   「アプリケーションを選択する」ページで、[ウェブアプリケーション] を選択します。
6. テストに適切な名前を付けます。これには、スペースまたは句読点の任意の組み合わせを含めることができます。

7. [次へ を選択します。
8. 「構成する」ページの [テストフレームワークをセットアップ] セクションにある [Appium **##**]を選択して、[ファイルを選択] を選択します。
9. テストが含まれている .zip ファイルを参照して選択します。この .zip ファイルは「Appium テストパッケージを構成する」で説明されている形式に従う必要があります。
10. [カスタム環境でテストを実行] を選択します。この実行環境を使用すると、テストの設定、ティアダウン、呼び出しを完全に制御できるとともに、ランタイムと Appium サーバーの特定バージョンを選択できます。テスト仕様ファイルを使用してカスタム環境を設定できます。詳細については、「AWS Device Farm のカスタムテスト環境による作業」を参照してください。
11. [次へ] を選択し、手順に従ってデバイスを選択して、実行を開始します。詳細については、「Device Farm でのテスト実行の作成」を参照してください。

> **Note**
>
> Device Farm で Appium テストは変更されません。

テストのスクリーンショットを撮る (オプション)

テストの一部としてスクリーンショットを撮影できます。

Device Farm は、`DEVICEFARM_SCREENSHOT_PATH` プロパティをローカルファイルシステム上の完全修飾パスに設定します。Device Farm は、そこを Appium スクリーンショットの保存先とみなします。スクリーンショットが保存されているテスト固有のディレクトリは、実行時に定義されます。スクリーンショットは Device Farm レポートに自動的に取り込まれます。スクリーンショットを表示するには、Device Farm コンソールで、[スクリーンショット] セクションを選択します。

Appium テストでのスクリーンショットの撮影の詳細については、Appium API ドキュメントの「スクリーンショットを撮る」を参照してください。

AWS Device Farm での Android テストによる作業

Device Farm では、Android デバイスの複数の自動化テストタイプ、および 2 種類のビルトインテストがサポートされています。

Android アプリケーションテストフレームワーク

Android デバイスでは、次のテストを使用できます。

- Appium と AWS Device Farm による作業
- Android および AWS Device Farm のインストゥルメンテーションによる作業

Android 用ビルトインテストタイプ

1 つのビルトインのテストタイプが Android デバイスで利用できます。

- ビルトイン: ファズ (Android および iOS)

Android および AWS Device Farm のインストゥルメンテーションによる作業

Device Farm では、Android 用のインストゥルメンテーション (JUnit、Espresso、Robotium、または実装ベースのテスト) のサポートを提供します。

Device Farm には、サンプルの Android アプリケーションと、インストゥルメンテーション (Espresso) を含む 3 つの Android オートメーションフレームワークでの動作テストへのリンクが用意されています。Android 用 Device Farm サンプルアプリケーションは、 でダウンロードできます GitHub。

トピック

- インストゥルメンテーションについて
- Android インストルメンテーションテストをアップロードする
- Android インストゥルメンテーションテストでのスクリーンショットの撮影
- Android インストゥルメンテーションテストに関するその他の考慮事項
- スタンダードモードのテスト解析

インストゥルメンテーションについて

Android のインストゥルメンテーションはテストコードでコールバックメソッドを呼び出すことができます。これにより、コンポーネントをデバッグしているかのように、コンポーネントのライフサイ

クルを段階的に実行できます。詳細については、「Android 開発者ツール」ドキュメントの「テストのタイプと場所」セクション内の「インストルメント化テスト」を参照してください。

Android インストルメンテーションテストをアップロードする

Device Farm コンソールを使用してテストをアップロードします。

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. プロジェクトのリストで、テストをアップロードするプロジェクトを選択します。

> **Tip**
>
> 検索バーで名前によりプロジェクトリストを絞り込めます。
> プロジェクトを作成するには、「AWS Device Farm でプロジェクトを作成する」の手順に従ってください。

4. [新規実行を作成] ボタンが表示されている場合は、選択します。
5. 「アプリケーションを選択する」ページで、[ファイルを選択] を選びます。
6. Android アプリケーションファイルを参照して選択します。このファイルは、.apk ファイルである必要があります。
7. [次へ] を選択します。
8. 「構成する」ページの [テストフレームワークをセットアップ] セクションにある [インストルメンテーション] を選択して、[ファイルを選択] を選びます。
9. テストが含まれている .apk ファイルを参照して選択します。
10. [次へ] を選択し、残りの手順を完了してデバイスを選択し、実行を開始します。

Android インストゥルメンテーションテストでのスクリーンショットの撮影

Android インストゥルメンテーションインストゥルメンテーションテストの一部としてスクリーンショットを撮ることができます。

スクリーンショットを撮るには、次のいずれかのメソッドを呼び出します:

- Robotium の場合は、`takeScreenShot` メソッドを呼び出します (例: `solo.takeScreenShot();`)。

- Spoon の場合は、次のような `screenshot` メソッドを呼び出します:

```
Spoon.screenshot(activity, "initial_state");
/* Normal test code... */
Spoon.screenshot(activity, "after_login");
```

テスト実行中、Device Farm は、デバイス上の次の場所 (存在する場合) からスクリーンショットを撮影し、テストレポートに追加します:

- `/sdcard/robotium-screenshots`
- `/sdcard/test-screenshots`
- `/sdcard/Download/spoon-screenshots/`*`test-class-name`*`/`*`test-method-name`*
- `/data/data/`*`application-package-name`*`/app_spoon-screenshots/`*`test-class-name`*`/`*`test-method-name`*

Android インストゥルメンテーションテストに関するその他の考慮事項

システムアニメーション

「Espresso テスト用 Android ドキュメント」に基づき、実際のデバイスでテストするときはシステムアニメーションをオフにすることをお勧めします。Device Farm は、android.support.test.runner.AndroidJ インストルメンテーションテストランナーで実行すると、ウィンドウアニメーションスケール 、移行アニメーションスケール 、およびアニメーション持続時間スケールの設定を自動的に無効にします。 AndroidJUnitRunner

テストレコーダー

Device Farm は、 record-and-playback スクリプティングツールを備えた Robotium などのフレームワークをサポートしています。

スタンダードモードのテスト解析

実行の標準モードでは、Device Farm はテストスイートを解析し、実行する固有のテストクラスおよびメソッドを識別します。これは Dex Test Parser というツールを使って行われます。

Android インストゥルメンテーションの .apk ファイルを入力として指定すると、パーサーは JUnit 3 および JUnit 4 コンベンションに一致するテストの完全修飾メソッド名を返します。

これをローカル環境でテストするには:

1. dex-test-parser バイナリーをダウンロードします。
2. 次のコマンドを実行して、Device Farm で実行されるテストメソッドのリストを取得します:

```
java -jar parser.jar path/to/apk path/for/output
```

AWS Device Farm での iOS テストによる作業。

Device Farm では、iOS デバイスの複数の自動化テストタイプ、および、ビルトインテストがサポートされています。

iOS アプリケーションテストフレームワーク

iOS デバイスでは、次のテストを使用できます。

- Appium と AWS Device Farm による作業
- iOS 用 XCTest と AWS Device Farm による作業
- XCTest UI

iOS 用ビルトインテストタイプ

現在、iOS デバイスで 1 つの組み込みのテストタイプが利用できます。

- ビルトイン: ファズ (Android および iOS)

iOS 用 XCTest と AWS Device Farm による作業

Device Farm により、XCTest フレームワークを使用してアプリケーションを実際のデバイスでテストできます。XCTest の詳細については、「Xcode によるテスト」の 「テストの基本」を参照してください。

テストを実行するには、テスト実行用のパッケージを作成し、これらのパッケージを Device Farm にアップロードします。

トピック

XCTest 実行用のパッケージの作成

XCTest フレームワークによりアプリケーションをテストするには、Device Farm に以下が必要です:

- `.ipa` ファイルのアプリケーションパッケージ。
- `.zip` ファイルの XCTest パッケージ。

Xcode が生成するビルド出力を使用してこれらのパッケージを作成します。次のステップを完了してパッケージを作成し、Device Farm にアップロードできるようにします。

アプリケーションのビルド出力を生成するには

1. Xcode でアプリケーションプロジェクトを開きます。
2. Xcode ツールバーのスキームのドロップダウンメニューで、[汎用 iOS デバイス] を送信先として選択します。
3. [製作物] メニューで、[ビルド用途] を選択した後、[テスト] を選択します。

アプリケーションパッケージを作成するには

1. Xcode のプロジェクトナビゲーターの [製作物] で、*app-project-name*`.app` という名前のファイルのコンテキストメニューを開きます。次に、[Finder で表示] を選択します。`Debug-iphoneos` という名前のフォルダが Finder で開きます。ここに、Xcode によってテストビルド用に生成された出力が含まれています。このフォルダには `.app` ファイルが含まれています。
2. Finder で、新規フォルダを作成して `Payload` という名前を付けます。
3. *app-project-name*`.app` ファイルをコピーして、`Payload` フォルダに貼り付けます。
4. `Payload` フォルダのコンテキストメニューを開き、[「Payload」を圧縮] を選択します。`Payload.zip` という名前のファイルが作成されます。
5. `Payload.zip` のファイル名と拡張子を *app-project-name*`.ipa` に変更します。

   後のステップで、このファイルを Device Farm に提供します。ファイルは、見つけやすくするためにデスクトップなど別の場所に移動させても構いません。
6. `Payload` フォルダとその中にある `.app` ファイルは必要に応じて削除できます。

XCTest パッケージを作成するには

1. Finder を使用し、Debug-iphoneos ディレクトリで *app-project-name*.app ファイルのコンテキストメニューを開きます。次に、[パッケージ内容を表示] を選択します。
2. パッケージ内容の中で、Plugins フォルダを開きます。 このフォルダに *app-project-name*.xctest という名前のファイルが含まれています。
3. このファイルのコンテキストメニューを開き、[「***app-project-name*.xctest**」を圧縮] を選択します。*app-project-name*.xctest.zip という名前のファイルが作成されます。

   後のステップで、このファイルを Device Farm に提供します。ファイルは、見つけやすくするためにデスクトップなど別の場所に移動させても構いません。

Device Farm への XCTest 実行用パッケージのアップロード

Device Farm コンソールを使用してテスト用パッケージをアップロードします。

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. まだプロジェクトがない場合は作成します。プロジェクトを作成するステップについては、「AWS Device Farm でプロジェクトを作成する」を参照してください。

   それ以外の場合は、Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. テストを実行するために使用するプロジェクトを選択します。
4. [新規実行を作成] を選択します。
5. 「アプリケーションを選択する」ページで、[モバイルアプリケーション] を選択します。
6. [ファイルを選択] を選択します。
7. アプリケーション用の .ipa ファイルを見つけ、アップロードします。

   > **Note**
   >
   > .ipa パッケージはテスト用にビルドされている必要があります。

8. アップロードが完了したら、[次へ] を選択します。
9. 「構成する」ページの [テストフレームワークをセットアップ] セクションで、[XCTest] を選択します。次に、[ファイルを選択] を選びます。

10. アプリケーション用 XCTest パッケージが含まれている .zip ファイルを見つけてアップロードします。
11. アップロードが完了したら、[次へ] を選択します。
12. プロジェクトの作成プロセスの残りのステップを完了します。テストするデバイスを選択し、デバイス状態を指定します。
13. 実行を構成したら、「実行を確認して開始する」ページで、[実行を確認して開始] を選択します。

    Device Farm によってテストが実行され、結果がコンソールに表示されます。

iOS 用 XCTest UI テストフレームワークと AWS Device Farm による作業

Device Farm では、iOS 用 XCTest UI テストフレームワークのサポートを提供しています。中でも、Device Farm では、Objective-C と Swift の両方で記述される XCTest UI テストをサポートしています。

トピック

- XCTest UI テストフレームワークについて
- iOS XCTest UI テストを準備する
- iOS XCTest UI テストをアップロードする
- iOS XCTest UI テストのスクリーンショットの撮影

XCTest UI テストフレームワークについて

XCTest UI フレームワークは Xcode 7 で導入された新規テストフレームワークです。このフレームワークは、UI テスト機能を備えた XCTest を拡張します。詳細については、iOS 開発者ライブラリの「ユーザーインターフェイスのテスト」を参照してください。

iOS XCTest UI テストを準備する

iOS XCTest UI テストランナーバンドルは、適切にフォーマットされた .ipa ファイルに含まれている必要があります。

.ipa ファイルを作成するには、 my-project-nameUITest -Runner.app バンドルを空の Payload ディレクトリに配置します。次に、Payload ディレクトリを .zip ファイルにアーカイブし、ファイル拡張子を .ipa に変更します。*UITest-Runner.app バンドルは、テストのためにプロジェクトをビルドするときに Xcode によって生成されます。プロジェクトの Products ディレクトリにあります。

iOS XCTest UI テストをアップロードする

Device Farm コンソールを使用してテストをアップロードします。

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. プロジェクトのリストで、テストをアップロードするプロジェクトを選択します。

> **Tip**
>
> 検索バーで名前によりプロジェクトリストを絞り込めます。
> プロジェクトを作成するには、「AWS Device Farm でプロジェクトを作成する」の手順に従ってください

4. [新規実行を作成] ボタンが表示されている場合は、選択します。
5. 「アプリケーションを選択する」ページで、[ファイルを選択] を選びます。
6. iOS アプリケーションファイルを参照して選択します。このファイルは、.ipa ファイルである必要があります。

> **Note**
>
> .ipa ファイルがシミュレーター用ではなく iOS デバイス用に作成されていることを確認します。

7. [次へ] をクリックします。
8. 「構成する」ページの [テストフレームワークをセットアップ] セクションにある [XCTest UI] を選択して、[ファイルを選択] を選びます。
9. iOS XCTest UI テストランナーを含む .ipa ファイルを参照して選択します。
10. [次へ] を選択後、残りの画面上の指示を完了してテストを実行するデバイスを選択し、実行を開始します。

iOS XCTest UI テストのスクリーンショットの撮影

XCTest UI テストでは、テストのステップごとに自動的にスクリーンショットをキャプチャします。これらのスクリーンショットは、Device Farm テストレポートに表示されます。追加のコードは不要です。

AWS Device Farm でのウェブアプリケーションテストによる作業

Device Farm は、ウェブアプリケーション用に Appium によるテストを提供します。Device Farm での Appium テストの設定についての詳細は、「the section called “Appium”」を参照してください。

計測および計測対象外デバイスのルール

ここでいう計測とは、デバイスに対する請求を指します。デフォルトでは、Device Farm デバイスが計測され、無料試用期間が経過すると 1 分ごとに課金されます。また、計測対象外のデバイスを購入することもできます。これにより、毎月の定額料金で無制限のテストが可能になります。料金の詳細については、「AWS Device Farm 料金表」を参照してください。

iOS デバイスと Android デバイスの両方を含むデバイスプールで実行を開始する場合は、計測対象デバイスと計測対象外デバイスのルールがあります。例えば、計測対象外 Android デバイスが 5 個、計測対象外 iOS デバイスが 5 個ある場合、ウェブテスト実行では、計測対象外デバイスが使用されます。

別の例として、計測対象外 Android デバイスが 5 個あり、計測対象外 iOS デバイスはないとします。ウェブ実行のために Android デバイスのみを選択すると、計測対象外デバイスが使用されます。ウェブ実行のために Android デバイスと iOS デバイスの両方を選択すると、課金方法が計測され、計測対象外デバイスは使用されません。

AWS Device Farm でのビルトインテストによる作業

Device Farm では、Android デバイスおよび iOS デバイス用のビルトインテストタイプのサポートを提供します。

ビルトインテストタイプ

ビルトインテストを使用すると、スクリプトを記述することなくアプリケーションをテストできます。

- ビルトイン: ファズ (Android および iOS)

Device Farm のビルトインファズテストによる作業

Device Farm はビルトインファズテストタイプを提供します。

ビルトインファズテストについて

ビルトインファズテストでは、ユーザーインターフェイスイベントをデバイスにランダムに送信して、その結果をレポートします。

ビルトインファズテストタイプを使用する

Device Farm コンソールを使用して、ビルトインファズテストを実行します。

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. プロジェクトの一覧で、ビルトインファズテストを実行するプロジェクトを選択します。

   > **Tip**
   >
   > 検索バーで名前によりプロジェクトリストを絞り込むことができます。
   > プロジェクトを作成するには、「AWS Device Farm でプロジェクトを作成する」の手順に従ってください。

4. [新規実行を作成] ボタンが表示されている場合は、選択します。
5. 「アプリケーションを選択する」ページで、[ファイルを選択] を選びます。
6. ビルトインファズテストを実行するアプリケーションファイルを参照して選択します。
7. 次へ をクリックします。
8. 「構成する」ページの [テストフレームワークをセットアップ] セクションで [ビルトイン: ファズ] を選択します。
9. 以下のいずれかの設定が表示された場合は、デフォルト値をそのまま使用するか、独自の値を指定できます:

   - イベント数: ファズテストが実行するユーザーインターフェイスイベントの数を 1 ~ 10,000 の範囲で指定します。
   - イベント調整: 次のユーザーインターフェイスイベントを実行する前にファズテストが待機するミリ秒数を 0 ~ 1,000 の範囲で指定します。
   - ランダマイザーシード: ファズテストがユーザーインターフェイスイベントのランダム化に使用する数を指定します。後続のファズテストに同じ番号を指定すると、同じイベントシーケンスが確保されます。

10. [次へ] を選択して残りの手順を完了し、デバイスを選択して実行を開始します。

AWS Device Farm のカスタムテスト環境による作業

AWS Device Farm では、自動テスト (カスタムモード) 用のカスタム環境を構成できます。これは、すべての Device Farm ユーザーに推奨される方法です。Device Farm の環境について詳しくは、「テスト環境」を参照してください。

標準モードとは対照的なカスタムモードの利点は次のとおりです:

- end-to-end テスト実行の高速化: テストパッケージはスイート内のすべてのテストを検出するために解析されないため、前処理/後処理のオーバーヘッドを回避できます。
- ライブログとビデオストリーミング: カスタムモードを使用すると、クライアント側のテストログとビデオがライブストリーミングされます。この機能は、標準モードでは使用できません。
- すべてのアーティファクトをキャプチャ: ホストとデバイスのカスタムモードではすべてのテストアーティファクトをキャプチャできます。この処理は、標準モードでは不可能な場合があります。
- より一貫性が高く複製可能なローカル環境: 標準モードでは、個々のテストごとにアーティファクトが個別に提供されるため、特定の状況下では有益な場合があります。ただし、Device Farm は実行された各テストを異なる方法で処理するため、ローカルテスト環境が元の構成と異なる場合があります。

  対照的に、カスタムモードでは、Device Farm のテスト実行環境をローカルテスト環境と一貫性のあるものにできます。

カスタム環境は、YAML 形式のテスト仕様ファイルを使用して構成されます。Device Farm には、サポートされているテストタイプごとにデフォルトのテスト仕様ファイルが用意されており、そのまま使用できますが、テストフィルタや構成ファイルなどをカスタマイズしてテスト仕様に追加することもできます。編集したテスト仕様は、future テスト実行のために保存できます。

詳細については、「AWS CLI を使用したカスタムテスト仕様のアップロード」および「Device Farm でのテスト実行の作成」を参照してください。

トピック

- テスト仕様構文
- テスト仕様の例。
- Android テスト用 Amazon Linux 2 テスト環境による作業
- 環境変数
- 標準テスト環境からカスタムテスト環境へのテストの移行

- Device Farm でのカスタムテスト環境の拡張

テスト仕様構文

これはYAML テスト仕様ファイルの構造を表します:

```
version: 0.1

phases:
  install:
    commands:
      - command
      - command
  pre_test:
    commands:
      - command
      - command
  test:
    commands:
      - command
      - command
  post_test:
    commands:
      - command
      - command

artifacts:
  - location
  - location
```

テスト仕様には次のものが含まれています:

`version`

Device Farm でサポートされているテスト仕様バージョンが反映されます。現在のバージョン番号は 0.1 です。

`phases`

このセクションは、テスト実行中に実行されるコマンドのグループを含みます。

許可されるテストフェーズ名は次のとおりです:

install

オプション。

Device Farm によってサポートされるテストフレームワークの依存関係はデフォルトで既にインストールされています。このフェーズには、追加コマンドが含まれます (インストール時に Device Farm で実行するコマンドがある場合)。

pre_test

オプション。

自動テスト実行前に実行されたコマンド (ある場合)。

test

オプション。

自動テスト実行中に実行されたコマンド。テストフェーズでいずれかのコマンドが失敗した場合、そのテストは失敗としてマークされます。

post_test

オプション。

自動テスト実行後に実行されたコマンド (ある場合)。

artifacts

オプション。

Device Farm は、カスタムレポート、ログファイル、画像などのアーティファクトをここで指定した場所から収集します。ワイルドカード文字はアーティファクトの場所の一部としてサポートされていないため、場所ごとに有効なパスを指定する必要があります。

これらのテストアーティファクトは、テスト実行でデバイスごとに表示されます。テストアーティファクトの取得については、「カスタムテスト環境でのアーティファクトの使用」を参照してください。

> **Important**
>
> テスト仕様は、有効な YAML ファイルとしてフォーマットされる必要があります。テスト仕様のインデントまたはスペースが無効の場合は、テスト実行が失敗する可能性があります。タブは YAML ファイルでは使用できません。テスト仕様が有効な YAML かどうかをテスト

するには、YAML バリデーターを使用します。詳細については、「YAML ウェブサイト」を参照してください。

テスト仕様の例。

これは、Appium Java TestNG テスト実行を構成する Device Farm YAML テスト仕様の一例です:

```
version: 0.1

# This flag enables your test to run using Device Farm's Amazon Linux 2 test host when
 scheduled on
# Android devices. By default, iOS device tests will always run on Device Farm's macOS
 test hosts.
# For Android, you can explicitly select your test host to use our Amazon Linux 2
 infrastructure.
# For more information, please see:
# https://docs.aws.amazon.com/devicefarm/latest/developerguide/amazon-linux-2.html
android_test_host: amazon_linux_2

# Phases represent collections of commands that are executed during your test run on
 the test host.
phases:

  # The install phase contains commands for installing dependencies to run your tests.
  # For your convenience, certain dependencies are preinstalled on the test host.

  # For Android tests running on the Amazon Linux 2 test host, many software libraries
 are available
  # from the test host using the devicefarm-cli tool. To learn more, please see:
  # https://docs.aws.amazon.com/devicefarm/latest/developerguide/amazon-linux-2-
devicefarm-cli.html

  # For iOS tests, you can use the Node.JS tools nvm, npm, and avm to setup your
 environment. By
  # default, Node.js versions 16.20.2 and 14.19.3 are available on the test host.
  install:
    commands:
      # The Appium server is written using Node.js. In order to run your desired
 version of Appium,
      # you first need to set up a Node.js environment that is compatible with your
 version of Appium.
```

```
    - |-
      if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "Android" ];
      then
        devicefarm-cli use node 16;
      else
        # For iOS, use "nvm use" to switch between the two preinstalled NodeJS
versions 14 and 16,
        # and use "nvm install" to download a new version of your choice.
        nvm use 16;
      fi;
    - node --version

    # Use the devicefarm-cli to select a preinstalled major version of Appium on
Android.
    # Use avm or npm to select Appium for iOS.
    - |-
      if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "Android" ];
      then
        # For Android, the Device Farm service automatically updates the preinstalled
Appium versions
        # over time to incorporate the latest minor and patch versions for each major
version. If you
        # wish to select a specific version of Appium, you can instead use NPM to
install it:
        # npm install -g appium@2.1.3;
        devicefarm-cli use appium 2;
      else
        # For iOS, Appium versions 1.22.2 and 2.2.1 are preinstalled and selectable
through avm.
        # For all other versions, please use npm to install them. For example:
        # npm install -g appium@2.1.3;
        # Note that, for iOS devices, Appium 2 is only supported on iOS version 14
and above using
        # NodeJS version 16 and above.
        avm 2.2.1;
      fi;
    - appium --version

    # For Appium version 2, for Android tests, Device Farm automatically updates the
preinstalled
    # UIAutomator2 driver over time to incorporate the latest minor and patch
versions for its major
    # version 2. If you want to install a specific version of the driver, you can use
the Appium
```

```
      # extension CLI to uninstall the existing UIAutomator2 driver and install your
desired version:
      # - |-
      #   if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "Android" ];
      #   then
      #     appium driver uninstall uiautomator2;
      #     appium driver install uiautomator2@2.34.0;
      #   fi;

      # For Appium version 2, for iOS tests, the XCUITest driver is preinstalled using
version 5.7.0
      # If you want to install a different version of the driver, you can use the
Appium extension CLI
      # to uninstall the existing XCUITest driver and install your desired version:
      # - |-
      #   if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "iOS" ];
      #   then
      #     appium driver uninstall xcuitest;
      #     appium driver install xcuitest@5.8.1;
      #   fi;

      # We recommend setting the Appium server's base path explicitly for accepting
commands.
      - export APPIUM_BASE_PATH=/wd/hub

      # Install the NodeJS dependencies.
      - cd $DEVICEFARM_TEST_PACKAGE_PATH
      # First, install dependencies which were packaged with the test package using
npm-bundle.
      - npm install *.tgz
      # Then, optionally, install any additional dependencies using npm install.
      # If you do run these commands, we strongly recommend that you include your
package-lock.json
      # file with your test package so that the dependencies installed on Device Farm
match
      # the dependencies you've installed locally.
      # - cd node_modules/*
      # - npm install

  # The pre-test phase contains commands for setting up your test environment.
  pre_test:
    commands:
      # Device farm provides different pre-built versions of WebDriverAgent, an
essential Appium
```

```
      # dependency for iOS devices, and each version is suggested for different
 versions of Appium:
      # DEVICEFARM_WDA_DERIVED_DATA_PATH_V8: this version is suggested for Appium 2
      # DEVICEFARM_WDA_DERIVED_DATA_PATH_V7: this version is suggested for Appium 1
      # Additionally, for iOS versions 16 and below, the device unique identifier
 (UDID) needs
      # to be slightly modified for Appium tests.
      - |-
        if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "iOS" ];
        then
          if [ $(appium --version | cut -d "." -f1) -ge 2 ];
          then
            DEVICEFARM_WDA_DERIVED_DATA_PATH=$DEVICEFARM_WDA_DERIVED_DATA_PATH_V8;
          else
            DEVICEFARM_WDA_DERIVED_DATA_PATH=$DEVICEFARM_WDA_DERIVED_DATA_PATH_V7;
          fi;

          if [ $(echo $DEVICEFARM_DEVICE_OS_VERSION | cut -d "." -f 1) -le 16 ];
          then
            DEVICEFARM_DEVICE_UDID_FOR_APPIUM=$(echo $DEVICEFARM_DEVICE_UDID | tr -d
 "-");
          else
            DEVICEFARM_DEVICE_UDID_FOR_APPIUM=$DEVICEFARM_DEVICE_UDID;
          fi;
        fi;

      # Appium downloads Chromedriver using a feature that is considered insecure for
 multitenant
      # environments. This is not a problem for Device Farm because each test host is
 allocated
      # exclusively for one customer, then terminated entirely. For more information,
 please see
      # https://github.com/appium/appium/blob/master/packages/appium/docs/en/guides/
security.md

      # We recommend starting the Appium server process in the background using the
 command below.
      # The Appium server log will be written to the $DEVICEFARM_LOG_DIR directory.
      # The environment variables passed as capabilities to the server will be
 automatically assigned
      # during your test run based on your test's specific device.
      # For more information about which environment variables are set and how they're
 set, please see
```

```
      # https://docs.aws.amazon.com/devicefarm/latest/developerguide/custom-test-
environment-variables.html
      - |-
        if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "Android" ];
        then
          appium --base-path=$APPIUM_BASE_PATH --log-timestamp \
            --log-no-colors --relaxed-security --default-capabilities \
            "{\"appium:deviceName\": \"$DEVICEFARM_DEVICE_NAME\", \
            \"platformName\": \"$DEVICEFARM_DEVICE_PLATFORM_NAME\", \
            \"appium:app\": \"$DEVICEFARM_APP_PATH\", \
            \"appium:udid\":\"$DEVICEFARM_DEVICE_UDID\", \
            \"appium:platformVersion\": \"$DEVICEFARM_DEVICE_OS_VERSION\", \
            \"appium:chromedriverExecutableDir\":
 \"$DEVICEFARM_CHROMEDRIVER_EXECUTABLE_DIR\", \
            \"appium:automationName\": \"UiAutomator2\"}" \
            >> $DEVICEFARM_LOG_DIR/appium.log 2>&1 &
        else
          appium --base-path=$APPIUM_BASE_PATH --log-timestamp \
            --log-no-colors --relaxed-security --default-capabilities \
            "{\"appium:deviceName\": \"$DEVICEFARM_DEVICE_NAME\", \
            \"platformName\": \"$DEVICEFARM_DEVICE_PLATFORM_NAME\", \
            \"appium:app\": \"$DEVICEFARM_APP_PATH\", \
            \"appium:udid\":\"$DEVICEFARM_DEVICE_UDID_FOR_APPIUM\", \
            \"appium:platformVersion\": \"$DEVICEFARM_DEVICE_OS_VERSION\", \
            \"appium:derivedDataPath\": \"$DEVICEFARM_WDA_DERIVED_DATA_PATH\", \
            \"appium:usePrebuiltWDA\": true, \
            \"appium:automationName\": \"XCUITest\"}" \
            >> $DEVICEFARM_LOG_DIR/appium.log 2>&1 &
        fi;

      # This code will wait until the Appium server starts.
      - |-
        appium_initialization_time=0;
        until curl --silent --fail "http://0.0.0.0:4723${APPIUM_BASE_PATH}/status"; do
          if [[ $appium_initialization_time -gt 30 ]]; then
            echo "Appium did not start within 30 seconds. Exiting...";
            exit 1;
          fi;
          appium_initialization_time=$((appium_initialization_time + 1));
          echo "Waiting for Appium to start on port 4723...";
          sleep 1;
        done;

  # The test phase contains commands for running your tests.
```

```
  test:
    commands:
      # Your test package is downloaded and unpackaged into the
 $DEVICEFARM_TEST_PACKAGE_PATH directory.
      # When compiling with npm-bundle, the test folder can be found in the
 node_modules/*/ subdirectory.
      - cd $DEVICEFARM_TEST_PACKAGE_PATH/node_modules/*
      - echo "Starting the Appium NodeJS test"

      # Enter your command below to start the tests. The command should be the same
 command as the one
      # you use to run your tests locally from the command line. An example, "npm
 test", is given below:
      - npm test

  # The post-test phase contains commands that are run after your tests have completed.
  # If you need to run any commands to generating logs and reports on how your test
 performed,
  # we recommend adding them to this section.
  post_test:
    commands:

# Artifacts are a list of paths on the filesystem where you can store test output and
 reports.
# All files in these paths will be collected by Device Farm.
# These files will be available through the ListArtifacts API as your "Customer
 Artifacts".
artifacts:
  # By default, Device Farm will collect your artifacts from the $DEVICEFARM_LOG_DIR
 directory.
  - $DEVICEFARM_LOG_DIR
```

Android テスト用 Amazon Linux 2 テスト環境による作業

AWS Device Farm は、Amazon Linux 2 を実行している Amazon Elastic Compute Cloud (EC2) ホストマシンを利用して Android テストを遂行します。テスト実行をスケジュールすると、Device Farm は各デバイスが個別にテストを実行するよう専有ホストを割り当てます。ホストマシンは、生成されたアーティファクトとともにテスト実行後に終了します。

Amazon Linux 2 テストホストは、以前の Ubuntu ベースのシステムに代わる最新の Android テスト環境です。テスト仕様ファイルを使用して、Android テストを Amazon Linux 2 環境で実行することを選択できます。

Amazon Linux 2 ホストにはいくつかの利点があります:

- より迅速で信頼性の高いテスト: 新しいテストホストは、従来のホストと比べてテスト速度が大幅に向上し、特にテスト開始時間が短縮されます。Amazon Linux 2 ホストでは、テスト中の安定性と信頼性も向上しています。
- 手動テスト用のリモートアクセスの強化: 最新のテストホストへのアップグレードと改善により、Android の手動テストにおけるレイテンシーの低下とビデオのパフォーマンスの向上がもたらされます。
- 標準ソフトウェアバージョン選択: Device Farm は、Appium フレームワークのバージョンだけでなく、テストホストでの主要なプログラミング言語サポートも標準化するようになりました。サポートされている言語 (現在は Java、Python、Node.js、Ruby) と Appium について、新しいテストホストは発売後すぐに長期安定版リリースを提供します。`devicefarm-cli` ツールによる一元的なバージョン管理により、フレームワーク全体で一貫性のあるテスト仕様ファイル開発が可能になります。

トピック

対応ソフトウェア

Amazon Linux 2 テストホストには、Device Farm のテストフレームワークをサポートするために必要な多くのソフトウェアライブラリがプリインストールされており、起動時にすぐにテスト環境を利用できます。その他の必要なソフトウェアについては、テスト仕様ファイルを変更して、テストパッケージからインストールしたり、インターネットからダウンロードしたり、VPC 内のプライベートソースにアクセスしたりできます (詳細については「VPC ENI 」を参照)。詳細については、「テスト仕様ファイルの例」を参照してください。

現在、ホストでは以下のソフトウェアバージョンを使用できます:

ソフトウェアライブラリ	ソフトウェアバージョン	テスト仕様ファイルで使用するコマンド

Python	3.8	`devicefarm-cli use python 3.8`
	3.9	`devicefarm-cli use python 3.9`
	3.10	`devicefarm-cli use python 3.10`
Java	8	`devicefarm-cli use java 8`
	11	`devicefarm-cli use java 11`
	17	`devicefarm-cli use java 17`
NodeJS	16	`devicefarm-cli use node 16`
	18	`devicefarm-cli use node 18`
Ruby	2.7	`devicefarm-cli use ruby 2.7`
	3.2	`devicefarm-cli use ruby 3.2`
Appium	1	`devicefarm-cli use appium 1`
	2	`devicefarm-cli use appium 2`

テストホストには、`pip` や `npm` といったパッケージマネージャー (それぞれ Python と Node.js に付属)、Appium などのツール用の依存物 (Appium UIAutomator2 ドライバーなど) など、各ソフトウェ

アバージョンで一般的に使用されるサポートツールも含まれています。これにより、サポートされているテストフレームワークと連携するのに必要なツールが確実に入手できます。

devicefarm-cli ツール

Amazon Linux 2 テストホストは、ソフトウェアバージョンを選択するために devicefarm-cli と呼ばれる標準化されたバージョン管理ツールを使用します。このツールは とは別の AWS CLI もので、Device Farm テストホストでのみ使用できます。devicefarm-cli を使用すると、テストホストにプリインストールされている任意のソフトウェアバージョンに切り替えることができます。これにより、Device Farm のテスト仕様ファイルを長期にわたって簡単に管理でき、将来ソフトウェアバージョンをアップグレードするための予測可能なメカニズムも得られます。

以下のスニペットは devicefarm-cli のhelpページを示しています:

```
$ devicefarm-cli help
 Usage: devicefarm-cli COMMAND [ARGS]

     Commands:
         help                          Prints this usage message.
         list                          Lists all versions of software configurable
                                       via this CLI.
         use <software> <version>      Configures the software for usage within the
                                       current shell's environment.
```

devicefarm-cli を使った例をいくつか見てみましょう。このツールを使用して、テスト仕様ファイルの Python バージョンを *3.10* から *3.9* に変更するには、次のコマンドを実行します:

```
$ python --version
Python 3.10.12
$ devicefarm-cli use python 3.9
$ python --version
Python 3.9.17
```

Appium のバージョンを *1* から *2* に変更するには:

```
$ appium --version
1.22.3
$ devicefarm-cli use appium 2
$ appium --version
2.1.2
```

> **Tip**
>
> ソフトウェアバージョンを選択すると、devicefarm-cli は、Python 用の pip や NodeJS 用の npm など、その言語のサポートツールも切り替えることに注意してください。

Android テストホストの選択

> **Warning**
>
> レガシー Android テストホストは、2024 年 10 月 21 日に利用できなくなります。非推奨のプロセスは複数の日付に分割されることに注意してください。
>
> - 2024 年 4 月 22 日、新しいアカウントのジョブはアップグレードされたテストホストに転送されます。
> - 2024 年 9 月 2 日、すべての新規または変更されたテスト仕様ファイルは、アップグレードされたテストホストをターゲットにする必要があります。
> - 2024 年 10 月 21 日をもって、ジョブはレガシーテストホストで実行できなくなります。
>
> 互換性の問題を防ぐため、テスト仕様ファイルをamazon_linux_2ホストに設定します。

Android テストの場合、Device Farm では Amazon Linux 2 テストホストを選択するために、テスト仕様ファイルに次のフィールドが必要です:

```
android_test_host: amazon_linux_2 | legacy
```

Amazon Linux 2 テストホストでテストを実行するには amazon_linux_2 を使用します:

```
android_test_host: amazon_linux_2
```

Amazon Linux 2 の利点について詳しくは、こちらをご覧ください。

Device Farm では、Android テストにはレガシーホスト環境の代わりに Amazon Linux 2 ホストを使用することを推奨しています。レガシー環境を使用したい場合は、legacy を使用してレガシーテストホストでテストを実行します:

```
android_test_host: legacy
```

デフォルトでは、テストホストを選択していないテスト仕様ファイルはレガシーテストホストで実行されます。

非推奨とされた構文

以下は、テスト仕様ファイルで Amazon Linux 2 を選択する際の廃止された構文です:

```
preview_features:
  android_amazon_linux_2_host: true
```

このフラグを使用している場合、テストは引き続き Amazon Linux 2 で実行されます。ただし、将来においてメンテナンスのオーバーヘッドを避けるため、`preview_features` フラグセクションを削除して新しい `android_test_host` フィールドに置き換えることを強くお勧めします。

> **Warning**
>
> テスト仕様ファイルで `android_test_host` と `android_amazon_linux_2_host` の両フラグを使用するとエラーが返されます。1 つだけ使用するようにしてください。`android_test_host` が推奨されます。

テスト仕様ファイルの例。

次のスニペットは、Android 用 Amazon Linux 2 テストホストを使用して Appium NodeJS テスト実行を構成する Device Farm テスト仕様ファイルの例です:

```
version: 0.1

# This flag enables your test to run using Device Farm's Amazon Linux 2 test host. For
 more information,
# please see https://docs.aws.amazon.com/devicefarm/latest/developerguide/amazon-
linux-2.html
android_test_host: amazon_linux_2

# Phases represent collections of commands that are executed during your test run on
 the test host.
phases:
```

```
  # The install phase contains commands for installing dependencies to run your tests.
  # For your convenience, certain dependencies are preinstalled on the test host. To
 lean about which
  # software is included with the host, and how to install additional software, please
 see:
  # https://docs.aws.amazon.com/devicefarm/latest/developerguide/amazon-linux-2-
supported-software.html

  # Many software libraries you may need are available from the test host using the
 devicefarm-cli tool.
  # To learn more about what software is available from it and how to use it, please
 see:
  # https://docs.aws.amazon.com/devicefarm/latest/developerguide/amazon-linux-2-
devicefarm-cli.html

  install:
    commands:
      # The Appium server is written using Node.js. In order to run your desired
 version of Appium,
      # you first need to set up a Node.js environment that is compatible with your
 version of Appium.
      - devicefarm-cli use node 18
      - node --version

      # Use the devicefarm-cli to select a preinstalled major version of Appium.
      - devicefarm-cli use appium 2
      - appium --version

      # The Device Farm service automatically updates the preinstalled Appium versions
 over time to
      # incorporate the latest minor and patch versions for each major version. If you
 wish to
      # select a specific version of Appium, you can use NPM to install it.
      # - npm install -g appium@2.1.3

      # For Appium version 2, Device Farm automatically updates the preinstalled
 UIAutomator2 driver
      # over time to incorporate the latest minor and patch versions for its major
 version 2. If you
      # want to install a specific version of the driver, you can use the Appium
 extension CLI to
      # uninstall the existing UIAutomator2 driver and install your desired version:
      # - appium driver uninstall uiautomator2
      # - appium driver install uiautomator2@2.34.0
```

```
      # We recommend setting the Appium server's base path explicitly for accepting
 commands.
      - export APPIUM_BASE_PATH=/wd/hub

      # Install the NodeJS dependencies.
      - cd $DEVICEFARM_TEST_PACKAGE_PATH
      # First, install dependencies which were packaged with the test package using
 npm-bundle.
      - npm install *.tgz
      # Then, optionally, install any additional dependencies using npm install.
      # If you do run these commands, we strongly recommend that you include your
 package-lock.json
      # file with your test package so that the dependencies installed on Device Farm
 match
      # the dependencies you've installed locally.
      # - cd node_modules/*
      # - npm install

  # The pre-test phase contains commands for setting up your test environment.
  pre_test:
    commands:

      # Appium downloads Chromedriver using a feature that is considered insecure for
 multitenant
      # environments. This is not a problem for Device Farm because each test host is
 allocated
      # exclusively for one customer, then terminated entirely. For more information,
 please see
      # https://github.com/appium/appium/blob/master/packages/appium/docs/en/guides/
security.md

      # We recommend starting the Appium server process in the background using the
 command below.
      # The Appium server log will be written to the $DEVICEFARM_LOG_DIR directory.
      # The environment variables passed as capabilities to the server will be
 automatically assigned
      # during your test run based on your test's specific device.
      # For more information about which environment variables are set and how they're
 set, please see
      # https://docs.aws.amazon.com/devicefarm/latest/developerguide/custom-test-
environment-variables.html
      - |-
        appium --base-path=$APPIUM_BASE_PATH --log-timestamp \
```

```
          --log-no-colors --relaxed-security --default-capabilities \
          "{\"appium:deviceName\": \"$DEVICEFARM_DEVICE_NAME\", \
          \"platformName\": \"$DEVICEFARM_DEVICE_PLATFORM_NAME\", \
          \"appium:app\": \"$DEVICEFARM_APP_PATH\", \
          \"appium:udid\":\"$DEVICEFARM_DEVICE_UDID\", \
          \"appium:platformVersion\": \"$DEVICEFARM_DEVICE_OS_VERSION\", \
          \"appium:chromedriverExecutableDir\":
\"$DEVICEFARM_CHROMEDRIVER_EXECUTABLE_DIR\", \
          \"appium:automationName\": \"UiAutomator2\"}" \
          >> $DEVICEFARM_LOG_DIR/appium.log 2>&amp;1 &amp;

      # This code will wait until the Appium server starts.
      - |-
        appium_initialization_time=0;
        until curl --silent --fail "http://0.0.0.0:4723${APPIUM_BASE_PATH}/status"; do
          if [[ $appium_initialization_time -gt 30 ]]; then
            echo "Appium did not start within 30 seconds. Exiting...";
            exit 1;
          fi;
          appium_initialization_time=$((appium_initialization_time + 1));
          echo "Waiting for Appium to start on port 4723...";
          sleep 1;
        done;

  # The test phase contains commands for running your tests.
  test:
    commands:
      # Your test package is downloaded and unpackaged into the
$DEVICEFARM_TEST_PACKAGE_PATH directory.
      # When compiling with npm-bundle, the test folder can be found in the
node_modules/*/ subdirectory.
      - cd $DEVICEFARM_TEST_PACKAGE_PATH/node_modules/*
      - echo "Starting the Appium NodeJS test"

      # Enter your command below to start the tests. The command should be the same
command as the one
      # you use to run your tests locally from the command line. An example, "npm
test", is given below:
      - npm test

  # The post-test phase contains commands that are run after your tests have completed.
  # If you need to run any commands to generating logs and reports on how your test
performed,
  # we recommend adding them to this section.
```

```
  post_test:
    commands:

# Artifacts are a list of paths on the filesystem where you can store test output and
 reports.
# All files in these paths will be collected by Device Farm.
# These files will be available through the ListArtifacts API as your "Customer
 Artifacts".
artifacts:
  # By default, Device Farm will collect your artifacts from the $DEVICEFARM_LOG_DIR
 directory.
  - $DEVICEFARM_LOG_DIR
```

Amazon Linux 2 テストホストへの移行

> **Warning**
>
> レガシー Android テストホストは、2024 年 10 月 21 日に利用できなくなります。非推奨のプロセスは複数の日付に分割されることに注意してください。
>
> - 2024 年 4 月 22 日、新しいアカウントのジョブはアップグレードされたテストホストに転送されます。
> - 2024 年 9 月 2 日、すべての新規または変更されたテスト仕様ファイルは、アップグレードされたテストホストをターゲットにする必要があります。
> - 2024 年 10 月 21 日をもって、ジョブはレガシーテストホストで実行できなくなります。
>
> 互換性の問題を防ぐため、テスト仕様ファイルを`amazon_linux_2`ホストに設定します。

既存のテストをレガシーホストから新しい Amazon Linux 2 ホストに移行するには、新しいテスト仕様ファイルを既存のテスト仕様ファイルを基に開発します。推奨される方法は、テストタイプに合った新しいデフォルトのテスト仕様ファイルから始めることです。次に、関連するコマンドを古いテスト仕様ファイルから新しいテスト仕様ファイルに移行し、古いファイルをバックアップとして保存します。これにより、既存のコードを再利用しながら、新しいホスト用に最適化されたデフォルト仕様を活用できます。そこでは、新しい環境にコマンドを適応させる際の参照用にレガシーテスト仕様を保持しつつ、テスト用に最適に構成された新しいホストの利点を最大限に活用できます。

以下の手順により、古いテスト仕様ファイルのコマンドを再利用しながら、新しい Amazon Linux 2 テスト仕様ファイルを作成できます:

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします:
2. 自動化テストを含む Device Farm プロジェクトに移動します。
3. プロジェクトで [新規テスト実行を作成] を選択します。
4. テストフレームワーク用に以前に使用したアプリとテストパッケージを選択します。
5. [カスタム環境でテストを実行] を選択します。
6. テスト仕様ドロップダウンメニューから、レガシーテストホストでのテストに現在使用しているテスト仕様ファイルを選択します。
7. このファイルの内容をコピーし、後で参照できるようにテキストエディターにローカルで貼り付けます。
8. テスト仕様のドロップダウンメニューで、テスト仕様の選択内容を最新のデフォルトテスト仕様ファイルに変更します。
9. [編集] を選択すると、テスト仕様編集インターフェイスが表示されます。テスト仕様ファイルの最初の数行で、新しいテストホストにすでにオプトインされていることがわかります:

```
android_test_host: amazon_linux_2
```

10. テストホストを選択する構文はこちらで、テストホスト間の主な違いはこちらで見直してください。
11. ステップ 6 でローカル保存されたテスト仕様ファイルから、新しいデフォルトテスト仕様ファイルにコマンドを選択的に追加して編集します。次に、[名前を付けて保存] を選択して新しい仕様ファイルを保存します。Amazon Linux 2 テストホストでのテスト実行をスケジュールできるようになりました。

新しいテストホストとレガシーテストホストの違い

Amazon Linux 2 テストホストを使用するためにテスト仕様ファイルを編集し、レガシーテストホストからテストを移す場合は、以下の主な環境の違いに注意してください:

- ソフトウェアバージョンの選択: 多くの場合、デフォルトのソフトウェアバージョンが変更されているため、これまでレガシーテストホストでソフトウェアバージョンを明示的に選択していなかった場合は、`devicefarm-cli` を使用して Amazon Linux 2 テストホストで今すぐ指定することが必要になるかもしれません。ほとんどのユースケースで、お客様はご使用のソフトウェアのバージョンを選択することが推奨されます。`devicefarm-cli` のソフトウェアバージョンを選択すると、予測可能で一貫した使用感が得られ、Device Farm がそのバージョンをテストホストから削除する予定がある場合は、大量の警告が表示されます。

さらに、nvm、pyenv、avm、rvm などのソフトウェア選択ツールは削除され、新しい devicefarm-cli ソフトウェア選択システムが採用されました。

- 使用可能なソフトウェアバージョン: 以前にプリインストールされていたソフトウェアの多くのバージョンが削除され、新規バージョンが多数追加されました。そのため、devicefarm-cli を使用してソフトウェアバージョンを選択する際は、必ずサポート対象バージョンリストにあるバージョンを選択してください。
- 絶対パスとしてレガシーホストのテスト仕様ファイルにハードコーディングされているファイルパスは、Amazon Linux 2 テストホストでは期待どおりに機能しない可能性があります。通常、テスト仕様ファイルには使用しないことをお勧めします。すべてのテスト仕様ファイルコードで相対パスと環境変数を使用することをお勧めします。さらに、テストに必要なバイナリのほとんどはホストの PATH にあるので、その名前 (appium など) だけで仕様ファイルからすぐに実行できることに注意してください。
- パフォーマンスデータ収集は、現時点で新しいテストホストではサポートされていません。
- オペレーティングシステムバージョン: レガシーテストホストは Ubuntu オペレーティングシステムをベースとしていましたが、新しいテストホストは Amazon Linux 2 をベースにしています。その結果、利用可能なシステムライブラリとシステムライブラリバージョンで多少の違いがあることに気付くかもしれません。
- Appium Java ユーザーの場合、新しいテストホストのクラスパスにはプリインストールされた JAR ファイルが含まれていませんが、以前のホストのクラスパスには TestNG フレームワーク用に (環境変数 $DEVICEFARM_TESTNG_JAR を通して) 含まれていました。お客様には、テストフレームワークに必要な JAR ファイルをテストパッケージ内にパッケージ化し、テスト仕様ファイルから $DEVICEFARM_TESTNG_JAR 変数のインスタンスを削除することをお勧めします。詳細については、「Appium と AWS Device Farm の取り扱い」を参照してください。
- Appium ユーザーの場合、Android 用 Chromedriver にアクセスできるようにする新しい方法が採用され、$DEVICEFARM_CHROMEDRIVER_EXECUTABLE 環境変数は削除されました。新しい環境変数 $DEVICEFARM_CHROMEDRIVER_EXECUTABLE_DIR を使用する例については、「デフォルトのテスト仕様ファイル」を参照してください。

> **Note**
>
> デフォルトのテスト仕様ファイルにある既存の Appium サーバーコマンドをそのまま使用することを強くお勧めします。

ソフトウェアの観点からテストホスト間の違いについてフィードバックや質問がある場合は、サポートケースを通じてサービスチームに連絡することをお勧めします。

環境変数

環境変数は、自動テストで使用する値を表します。これらの環境変数は、YAML ファイルやテストコードで使用することができます。カスタムテスト環境で、Device Farm は実行時間に環境変数を動的に入力します。

トピック

一般的な環境変数

Android テスト

このセクションでは、Device Farm でサポートされている Android プラットフォームテストで一般的なカスタム環境変数について説明します。

$DEVICEFARM_DEVICE_NAME

テストを実行するデバイスの名前。デバイスの一意のデバイス識別子 (UDID) を表します。

$DEVICEFARM_DEVICE_PLATFORM_NAME

デバイスのプラットフォーム名。Android または iOS。

$DEVICEFARM_DEVICE_OS_VERSION

デバイスの OS バージョン。

$DEVICEFARM_APP_PATH

テストが実行されているホストマシン上のモバイルアプリケーションへのパス。アプリケーションパスは、モバイルアプリケーションでのみ使用できます。

$DEVICEFARM_DEVICE_UDID

自動テストを実行中のモバイルデバイスの一意の識別子。

$DEVICEFARM_LOG_DIR

テスト実行中に生成されるログファイルへのパス。デフォルトでは、このディレクトリ内のすべてのファイルは ZIP ファイルにアーカイブされ、テスト実行後にアーティファクトとして使用できるようになります。

$DEVICEFARM_SCREENSHOT_PATH

テスト実行中にキャプチャされるスクリーンショットへのパス (ある場合)。

$DEVICEFARM_CHROMEDRIVER_EXECUTABLE_DIR

Appium のウェブテストやハイブリッドテストで使用するために必要な Chromedriver 実行ファイルが格納されているディレクトリの場所。

$ANDROID_HOME

Android SDK インストールディレクトリへのパス。

> **Note**
>
> ANDROID_HOME 環境変数は、Android 用 Amazon Linux 2 テストホストでのみ使用できます。

iOS テスト

このセクションでは、Device Farm でサポートされている iOS プラットフォームテストで一般的なカスタム環境変数について説明します。

$DEVICEFARM_DEVICE_NAME

テストを実行するデバイスの名前。デバイスの一意のデバイス識別子 (UDID) を表します。

$DEVICEFARM_DEVICE_PLATFORM_NAME

デバイスのプラットフォーム名。Android または iOS。

$DEVICEFARM_APP_PATH

テストが実行されているホストマシン上のモバイルアプリケーションへのパス。アプリケーションパスは、モバイルアプリケーションでのみ使用できます。

$DEVICEFARM_DEVICE_UDID

自動テストを実行中のモバイルデバイスの一意の識別子。

$DEVICEFARM_LOG_DIR

テスト実行中に生成されるログファイルへのパス。

$DEVICEFARM_SCREENSHOT_PATH

テスト実行中にキャプチャされるスクリーンショットへのパス (ある場合)。

Appium Java JUnit 環境変数

このセクションでは、カスタムテスト環境の Appium Java JUnit テストで使用される環境変数を示します。

$DEVICEFARM_TESTNG_JAR

TestNG .jar ファイルへのパス。

$DEVICEFARM_TEST_PACKAGE_PATH

テストパッケージの解凍された中身へのパス。

Appium Java TestNG 環境変数

このセクションでは、カスタムテスト環境の Appium Java TestNG テストで使用される環境変数について説明します。

$DEVICEFARM_TESTNG_JAR

TestNG .jar ファイルへのパス。

$DEVICEFARM_TEST_PACKAGE_PATH

テストパッケージの解凍された中身へのパス。

XCUITest 環境変数

$DEVICEFARM_XCUITESTRUN_FILE

Device Farm `.xctestun` ファイルへのパス。アプリケーションとテストパッケージから生成されます。

$DEVICEFARM_DERIVED_DATA_PATH

Device Farm xcodebuild 出力の予想されるパス。

標準テスト環境からカスタムテスト環境へのテストの移行

以下のガイドでは、標準テスト実行モードからカスタム実行モードに切り替える方法を説明します。移行には主に 2 つの異なる実行形式が含まれます:

1. 標準モード: このテスト実行モードは、主に、詳細なレポートと完全管理された環境をお客様に提供するために構築されています。
2. カスタムモード: このテスト実行モードは、より迅速なテスト実行、リフトアンドシフトによるローカル環境と同等の機能、およびライブビデオストリーミングを必要とするさまざまなユースケース向けに構築されています。

移行する際の考慮事項

このセクションでは、カスタムモードへの移行時に考慮すべき主な使用事例をいくつか挙げます:

1. 速度: 標準実行モードで Device Farm は、特定のフレームワークのパッケージ化手順によりパッケージ化とアップロードを行ったテストのメタデータを解析します。解析により、パッケージ内のテスト数が検出されます。その後、Device Farm は各テストを個別に実行し、各テストのログ、ビデオ、他の結果アーティファクトを個別に表示します。ただし、 end-to-end テストと結果アーティファクトの前処理と後処理がサービス終了にあるため、これはテスト実行時間の合計に着実に増加します。

   これとは対照的に、カスタム実行モードではテストパッケージが解析されません。つまり、テストや結果アーティファクトの前処理や後処理は最小限に抑えられます。これにより、合計 end-to-end 実行時間がローカルセットアップに近いものになります。テストは、ローカルマシンで実行した場合と同じ形式で実行されます。テストの結果はローカルで取得したものと同じで、ジョブ実行の終了時にダウンロードできます。
2. カスタマイズまたは柔軟性: 標準実行モードでは、テストパッケージを解析してテスト数を検出し、各テストを個別に実行します。テストが指定した順序で実行される保証はないことに注意してください。その結果、特定の実行順序を必要とするテストが期待どおりに動作しない場合があります。さらに、ホストマシン環境をカスタマイズしたり、特定方法でテストを実行するために必要な可能性のある構成ファイルを渡す方法もありません。

対照的に、カスタムモードでは、追加ソフトウェアのインストール、テストへのフィルターの受け渡し、構成ファイルの受け渡し、テスト実行設定の制御など、ホストマシン環境を構成できます。これは yaml ファイル (testspec ファイルとも呼ばれます) を使用して実現します。yaml ファイルは、シェルコマンドを追加することで変更できます。この yaml ファイルはシェルスクリプトに変換され、テストホストマシン上で実行されます。複数の yaml ファイルを保存し、実行をスケジュールする際に要件に応じて動的に 1 つを選択できます。

3. ライブビデオとロギング: 標準実行モードとカスタム実行モードの両方で、テスト用のビデオとログが表示されます。ただし、標準モードでは、テストが完了して初めて、テストのビデオと定義済みのログが取得されます。

   これとは対照的に、カスタムモードではテストのビデオのライブストリームとクライアント側ログを提供します。さらに、テストの最後にビデオや他のアーティファクトをダウンロードすることもできます。

4. 廃止予定: 以下のテストタイプは、2023 年 12 月末までに標準実行モードで廃止される予定です。

   - Appium (すべての言語)
   - Calabash
   - XCTest
   - UI 自動化
   - UI オートメーター
   - ウェブテスト
   - Built-in: エクスプローラー

   廃止されると、これらのフレームワークは標準モードでは使用できなくなります。上記のテストタイプでは、代わりにカスタムモードを使用できます。

> **Tip**
>
> ユースケースに上記の要素のうち少なくとも 1 つが含まれる場合は、カスタム実行モードに切り替えることを強くお勧めします。

移行手順

標準モードからカスタムモードに移行するには、次の操作を行います:

1. AWS Management Console にサインインして https://console.aws.amazon.com/devicefarm で Device Farm コンソールを開きます。
2. プロジェクトを選択し、新しい自動化実行を開始します。
3. アプリをアップロード (または web app を選択) し、テストフレームワークの種類を選択し、テストパッケージをアップロードします。次に「Choose your execution environment」パラメーターの下で「Run your test in a custom environment」へのオプションを選択します。
4. デフォルトでは、Device Farm のサンプルテスト仕様ファイルが表示され、表示および編集できます。このサンプルファイルは、カスタム環境モードでテストを試す際の出発点として使用できます。次に、コンソールからテストが正しく動作していることを確認したら、API、CLI、および、Device Farm とのパイプライン統合を変更し、テスト実行をスケジュールする際にこのテスト仕様ファイルをパラメータとして使用できます。テスト仕様ファイルを実行用パラメータとして追加する方法については、「API ガイド」の ScheduleRun API の「testSpecArn パラメータ」セクションを参照してください。

Appium フレームワーク

カスタムテスト環境において、Device Farm は、Appium フレームワークテストでの Appium 機能の挿入または上書きを行いません。テストの Appium 機能は、テスト仕様 YAML ファイルまたはテストコードで指定する必要があります。

Android 実装

Android 実装テストをカスタムテスト環境に移行する際、変更は必要ありません。

iOS XCUITest

iOS XCUITest テストをカスタムテスト環境に移行する際、変更は必要ありません。

Device Farm でのカスタムテスト環境の拡張

Device Farm のカスタムモードを使用すると、単なるテストスイート以上のものが実行できます。このセクションでは、テストスイートを拡張し、テストを最適化する方法を説明します。

PIN の設定

一部のアプリケーションでは、デバイスに PIN を設定することが必要です。Device Farm では、デバイスの PIN をネイティブに設定することはサポートされていません。ただし、これは次のことに注意すれば可能となります:

- デバイスで Android 8 以上を実行している必要があります。
- テスト完了後、PIN を削除する必要があります。

テストで PIN を設定するには、次に示すように、`pre_test` および `post_test` フェーズを使用して PIN を設定および削除します:

```
phases:
    pre_test:
      - # ... among your pre_test commands
      - DEVICE_PIN_CODE="1234"
      - adb shell locksettings set-pin "$DEVICE_PIN_CODE"
    post_test:
      - # ... Among your post_test commands
      - adb shell locksettings clear --old "$DEVICE_PIN_CODE"
```

テストスイートが開始されると、PIN 1234 が設定されます。テストスイートの終了後に、PIN が削除されます。

> **Warning**
>
> テストの完了後にデバイスから PIN を削除しないと、デバイスとアカウントが隔離されます。

必要な性能における Appium ベースのテストの高速化

Appium を使用すると、標準モードのテストスイートが非常に遅くなることがあります。これは、Device Farm がデフォルト設定を適用しており、ユーザーの希望する Appium 環境の使用方法について想定していないためです。これらのデフォルトは業界のベストプラクティスを中心にして構築されていますが、ご利用状況によっては適応しない場合があります。Appium サーバーのパラメーターを微調整するには、テスト仕様のデフォルト Appium 機能を調整してください。例えば、以下で

は、usePrebuildWDA 機能を iOS テストスイートで true に設定して、初期開始時間を高速化します:

```
phases:
  pre_test:
    - # ... Start up Appium
    - >-
    appium --log-timestamp
    --default-capabilities "{\"usePrebuiltWDA\": true, \"derivedDataPath\":
\"$DEVICEFARM_WDA_DERIVED_DATA_PATH\",
    \"deviceName\": \"$DEVICEFARM_DEVICE_NAME\", \"platformName\":
\"$DEVICEFARM_DEVICE_PLATFORM_NAME\", \"app\":\"$DEVICEFARM_APP_PATH\",
    \"automationName\":\"XCUITest\", \"udid\":\"$DEVICEFARM_DEVICE_UDID_FOR_APPIUM\",
 \"platformVersion\":\"$DEVICEFARM_DEVICE_OS_VERSION\"}"
    >> $DEVICEFARM_LOG_DIR/appiumlog.txt 2>&1 &
```

Appium の機能は、シェルエスケープされ、引用符で囲まれた JSON 構造でなければなりません。

次の Appium 機能は、パフォーマンス向上の一般的なソースとなります:

noReset および fullReset

これらの 2 つの機能は互いに排他的となっており、各セッションの完了後に Appium の動作を記述します。noReset が true に設定されている場合、Appium サーバーは Appium セッションが終了してもアプリケーションからデータを削除せず、実質的にいかなるクリーンアップも行いません。セッションが終了した後、fullReset が、デバイスからすべてのアプリケーションデータをアンインストールおよびクリアします。詳細については、Appium ドキュメントの「ストラテジーを初期化する」を参照してください。

ignoreUnimportantViews (Android のみ)

Appium に、テスト用の関連するビューにのみ Android UI 階層を圧縮するように指示し、ある特定要素の検索を高速化します。ただし、UI レイアウトの階層が変更されているため、XPath ベースのテストスイートの一部はこれによって壊れる可能性があります。

skipUnlock (Android のみ)

今設定されている PIN コードがないことを Appium に通知し、スクリーンオフイベントまたはその他のロックイベント後にテストを高速化します。

webDriverAgentUrl (iOS のみ)

重要な iOS 依存関係 webDriverAgent がすでに実行され、指定された URL で HTTP リクエストを受け付けることができると想定するように Appium に指示します。webDriverAgent がまだ起動していない場合、Appium がテストスイートの開始時に webDriverAgent を起動するまでに時間がかかることがあります。自分でwebDriverAgent を起動して Appium の起動時に webDriverAgentUrl を http://localhost:8100 に設定すると、テストスイートをより速く起動できます。この機能は useNewWDA 機能と一緒に使うべきではないことに注意してください。

次のコードを使用して、デバイスのローカルポート 8100 にあるテスト仕様ファイルから webDriverAgent を開始し、テストホストのローカルポート 8100 に転送できます (これにより、webDriverAgentUrl の値をhttp://localhost:8100 に設定できます)。このコードは、Appium と webDriverAgent 環境変数を設定するコードが定義されたあと、インストール段階で実行する必要があります:

```
      # Start WebDriverAgent and iProxy
      - >-
        xcodebuild test-without-building -project /usr/local/avm/versions/
$APPIUM_VERSION/node_modules/appium/node_modules/appium-webdriveragent/
WebDriverAgent.xcodeproj
        -scheme WebDriverAgentRunner -derivedDataPath
 $DEVICEFARM_WDA_DERIVED_DATA_PATH
        -destination id=$DEVICEFARM_DEVICE_UDID_FOR_APPIUM
 IPHONEOS_DEPLOYMENT_TARGET=$DEVICEFARM_DEVICE_OS_VERSION
        GCC_TREAT_WARNINGS_AS_ERRORS=0 COMPILER_INDEX_STORE_ENABLE=NO >>
 $DEVICEFARM_LOG_DIR/webdriveragent_log.txt 2>&1 &

        iproxy 8100 8100 >> $DEVICEFARM_LOG_DIR/iproxy_log.txt 2>&1 &
```

次に、テスト仕様ファイルに次のコードを追加して、webDriverAgent が正常に起動したことを確認できます。Appium が正常に起動したことを確認したら、テスト前のフェーズの最後にこのコードを実行する必要があります:

```
      # Wait for WebDriverAgent to start
      - >-
        start_wda_timeout=0;
        while [ true ];
        do
          if [ $start_wda_timeout -gt 60 ];
          then
```

```
            echo "WebDriverAgent server never started in 60 seconds.";
            exit 1;
        fi;
        grep -i "ServerURLHere" $DEVICEFARM_LOG_DIR/webdriveragent_log.txt >> /
dev/null 2>&1;
        if [ $? -eq 0 ];
        then
            echo "WebDriverAgent REST http interface listener started";
            break;
        else
            echo "Waiting for WebDriverAgent server to start. Sleeping for 1
 seconds";
            sleep 1;
            start_wda_timeout=$((start_wda_timeout+1));
        fi;
      done;
```

Appium がサポートする機能の詳細については、Appium ドキュメントの「Appium で必要な機能」を参照してください。

テスト実行後の Webhook と他の API の使用

curl を使用してすべてのテストスイートが終了した後は、Device Farm で Webhook を呼び出すことができます。これを行うプロセスは、宛先とフォーマットにより異なります。特定の Webhook については、その Webhook のドキュメンテーションを参照してください。次の例では、テストスイートの終了のたびにメッセージを Slack ウェブフックに投稿します:

```
phases:
  post_test:
    - curl -X POST -H 'Content-type: application/json' --data '{"text":"Tests on
 '$DEVICEFARM_DEVICE_NAME' have finished!"}' https://hooks.slack.com/services/
T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXXX
```

Slack で Webhook を使用することについての詳細は、Slack API リファレンスの「Webhook を使用した最初の Slack メッセージの送信」を参照してください。

あなたは Webhook の呼び出しに curl を使用することに制限されません。Device Farm 実行環境と互換性があるならば、テストパッケージには追加のスクリプトとツールを含ませられます。例えば、テストパッケージに、他の API にリクエストを行う補助スクリプトが含ませられる場合があります。要求したパッケージが、テストスイートの要件と一緒にインストールされていることを確認してくだ

さい。テストスイートの完了後に実行するスクリプトを追加するには、スクリプトをテストパッケージに含めて、テスト仕様に以下を追加します:

```
phases:
  post_test:
    - python post_test.py
```

> **Note**
>
> テストパッケージで使用される API キーまたは他の認証トークンの管理は、お客様の責任となります。あらゆる形式のセキュリティ認証情報をソース管理から除外するとともに、できるだけ少ない権限で認証情報を使用し、できる限り取り消し可能な短期間のトークンを使用することをお勧めします。セキュリティ要件を確認するには、使用するサードパーティ API のドキュメンテーションを参照してください。

テスト実行スイートの一部として AWS サービスを使用する予定の場合は、テストスイート外で生成され、テストパッケージに含まれている IAM 一時認証情報を使用する必要があります。これらの認証情報は可能な限り、与えられた権限数が少なく、存続期間が短い必要があります。一時的な認証情報の作成に関する詳細については、「IAM ユーザーガイド」の「一時的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。

テストパッケージへのファイルの追加

追加の構成ファイルまたは追加のテストデータとして、テストの一部として追加ファイルを使用することが必要な場合があります。これらの追加ファイルは、 AWS Device Farmにアップロードする前にテストパッケージに追加しておき、カスタム環境モードからアクセスできます。基本的に、すべてのテストパッケージのアップロード形式 (ZIP、IPA、APK、JAR など) は、標準 ZIP 操作をサポートするパッケージアーカイブ形式です。

次のコマンド AWS Device Farm を使用して、 にアップロードする前にテストアーカイブにファイルを追加できます。

```
$ zip zip-with-dependencies.zip extra_file
```

追加ファイルのディレクトリの場合:

```
$ zip -r zip-with-dependencies.zip extra_files/
```

これらのコマンドは、IPA ファイルを除くすべてのテストパッケージのアップロード形式で期待どおりに機能します。IPA ファイルの場合、特に XCUITests で使用する場合、 が iOS テストパッケージ AWS Device Farm を書き換える方法により、余分なファイルを少し別の場所に配置することをお勧めします。iOS テストをビルドするとき、テストアプリケーションディレクトリは *Payload* という名前の別のディレクトリ内にあります。

たとえば、このような iOS テストディレクトリは次のようになります:

```
$ tree
.
### Payload
    ### ADFiOSReferenceAppUITests-Runner.app
        ### ADFiOSReferenceAppUITests-Runner
        ### Frameworks
        #   ### XCTAutomationSupport.framework
        #   #   ### Info.plist
        #   #   ### XCTAutomationSupport
        #   #   ### _CodeSignature
        #   #   #   ### CodeResources
        #   #   ### version.plist
        #   ### XCTest.framework
        #       ### Info.plist
        #       ### XCTest
        #       ### _CodeSignature
        #       #   ### CodeResources
        #       ### en.lproj
        #       #   ### InfoPlist.strings
        #       ### version.plist
        ### Info.plist
        ### PkgInfo
        ### PlugIns
        #   ### ADFiOSReferenceAppUITests.xctest
        #   #   ### ADFiOSReferenceAppUITests
        #   #   ### Info.plist
        #   #   ### _CodeSignature
        #   #       ### CodeResources
        #   ### ADFiOSReferenceAppUITests.xctest.dSYM
        #       ### Contents
        #           ### Info.plist
        #           ### Resources
        #               ### DWARF
        #                   ### ADFiOSReferenceAppUITests
        ### _CodeSignature
```

```
        #   ### CodeResources
        ### embedded.mobileprovision
```

これらの XCUITest パッケージでは、*Payload* ディレクトリ内の *.app* で終わるディレクトリに余分なファイルを追加します。たとえば、以下のコマンドは、このテストパッケージにファイルを追加する方法を示しています:

```
$ mv extra_file Payload/*.app/
$ zip -r my_xcui_tests.ipa Payload/
```

テストパッケージにファイルを追加すると、アップロード形式に基づいて AWS Device Farm でインタラクションの動作が若干異なることが予想されます。ZIP ファイル拡張子を使用したアップロードでは、テスト前に AWS Device Farm がアップロードを自動的に解凍し、解凍したファイルを *$DEVICEFARM_TEST_PACKAGE_PATH* 環境変数のある場所に残します。(つまり、最初の例のように *extra_file* というファイルをアーカイブのルートに追加した場合、テスト中は *$DeviceFarm_Test_Package_path/Extra_File* に置かれることになります)。

より実用的な例を挙げると、Appium TestNG ユーザーがテストに *testng.xml* ファイルを含めたい場合、以下のコマンドを使用してアーカイブに含めることができます:

```
$ zip zip-with-dependencies.zip testng.xml
```

次に、カスタム環境モードのテストコマンドを次のように変更できます:

```
java -D appium.screenshots.dir=$DEVICEFARM_SCREENSHOT_PATH org.testng.TestNG -testjar
 *-tests.jar -d $DEVICEFARM_LOG_DIR/test-output $DEVICEFARM_TEST_PACKAGE_PATH/
testng.xml
```

テストパッケージのアップロード拡張子が ZIP でない場合 (APK、IPA、JAR ファイルなど)、アップロードされたパッケージファイル自体は *$DEVICEFARM_TEST_PACKAGE_PATH* にあります。これらはまだアーカイブ形式のファイルなので、ファイルを解凍すると、その中身から追加のファイルにアクセスできます。たとえば、次のコマンドはテストパッケージ (APK、IPA、または JAR ファイル用) の中身を */tmp* ディレクトリに解凍します。

```
unzip $DEVICEFARM_TEST_PACKAGE_PATH -d /tmp
```

APK または JAR ファイルの場合、追加ファイルは */tmp* ディレクトリ (例: */tmp/extra_file*) に解凍されます。IPA ファイルの場合、前に説明したように、余分なファイルは *Payload* ディレ

クトリ内の *.app* で終わるフォルダー内の少し異なる場所に配置されます。例えば、上記の IPA の例に基づいて、ファイルは */tmp/Payload/ADFiOSReferenceApp UITests -Runner.app/extra_file* (*/tmp/Payload/*.app/extra_file #######*) の場所にあります。

AWS Device Farm でのリモートアクセスによる作業

リモートアクセスでは、機能をテストして顧客の問題を再現するために、ウェブブラウザを使用してリアルタイムでデバイスのスワイプ、ジェスチャ、および操作を行えます。特定デバイスを操作するには、そのデバイスとのリモートアクセスセッションを作成します。

Device Farm でのセッションは、ウェブブラウザでホストされている実際の物理デバイスとのリアルタイムインタラクションとなります。セッションは、そのセッションを開始するときに選択した単一のデバイスを表示します。ユーザーは一度に複数セッションを開始できますが、同時に操作できるデバイスの総数は、お持ちのデバイススロットの数によって制限されます。デバイススロットは、デバイスファミリー (Android か iOS デバイス) に基づいて購入できます。詳細については、「Device Farm 料金表」を参照してください。

Device Farm では現在、リモートアクセステスト用のデバイスのサブセットを提供しています。デバイスプールには随時新しいデバイスが追加されます。

Device Farm は各リモートアクセスセッションのビデオをキャプチャし、セッション中にアクティビティのログを生成します。これらの結果には、セッション中に提供するすべての情報が含まれます。

> **Note**
>
> セキュリティ上の理由から、リモートアクセスセッション中に、アカウント番号、個人用ログイン情報、他の詳細などの機密情報を提供または入力しないことをお勧めします。

トピック

- AWS Device Farm でリモートアクセスセッションを作成する
- AWS Device Farm でリモートアクセスセッションを使用する
- AWS Device Farm でリモートアクセスセッションの結果を取得する

AWS Device Farm でリモートアクセスセッションを作成する

リモートアクセスセッションの詳細については、「セッション」を参照してください。

- 前提条件
- テスト実行を作成 (コンソール)

- 次のステップ

前提条件

- Device Farm でプロジェクトを作成します。「AWS Device Farm でプロジェクトを作成する」の指示に従ってから、このページに戻ります。

Device Farm コンソールによりセッションを作成する

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. 既にプロジェクトがある場合は、リストから選択します。それ以外の場合は、「AWS Device Farm でプロジェクトを作成する」の手順に従ってプロジェクトを作成します。
4. [リモートアクセス] タブで、[新規セッションを開始] を選択します。
5. セッション用のデバイスを選択します。使用可能なデバイスのリストから選択するか、リストの上部にある検索バーを使用してデバイスを検索します。次の方法で検索できます。
   - 名前
   - プラットフォーム
   - フォームファクタ
   - フリートタイプ
6. [セッション名] にセッションの名前を入力します。
7. [セッションを確認して開始] を選択します。

次のステップ

リクエストされたデバイスが利用可能になった時点 (通常は数分以内) で、Device Farm はセッションを開始します。セッションが開始するまで、「デバイスがリクエストされました」ダイアログボックスが表示されます。セッションリクエストをキャンセルするには、[リクエストをキャンセル] を選択します。

セッションが開始された後、セッションを停止せずにブラウザまたはブラウザのタブを閉じた場合、またはブラウザとインターネット間の接続が失われた場合、セッションは 5 分間アクティブなまま

維持されます。その後、Device Farm はセッションを終了します。アカウントにはアイドル時間に対しても課金されます。

セッションが開始したら、ウェブブラウザでデバイスとやり取りできます。

AWS Device Farm でリモートアクセスセッションを使用する

リモートアクセスセッションを通じた Android および iOS のアプリケーションのインタラクティブなテストの実行については、「セッション」を参照してください

- 前提条件
- Device Farm コンソールでセッションを使用する
- 次のステップ
- ヒントとコツ

前提条件

- セッションを作成します。「セッションを作成する」の指示に従ってから、このページに戻ります。

Device Farm コンソールでセッションを使用する

リモートアクセスセッションをリクエストしたデバイスが利用可能になるとすぐに、コンソールにデバイスの画面が表示されます。セッションの最大長は 150 分です。セッションの残り時間は、デバイス名の近くにある [残り時間] フィールドに表示されます。

アプリケーションのインストール

セッションデバイスにアプリケーションをインストールするには、[アプリケーションをインストール] で [ファイルを選択] を選び、インストールする .apk ファイル (Android) または .ipa ファイル (iOS) を選択します。リモートアクセスセッションで実行するアプリケーションは、テスト計測またはプロビジョニングを必要としません。

> **Note**
>
> アプリケーションのインストールが完了しても、AWS Device Farm に確認は表示されません。アプリケーションの使用準備ができたかどうか確認するには、アプリケーションアイコンを操作してみてください。

アプリケーションをアップロードするときに、アプリケーションが利用可能になるまでに遅延が発生することもあります。システムトレイを確認し、アプリケーションが利用可能かどうか調べます。

デバイスの制御

実際の物理デバイスと同じように、コンソールに表示されるデバイスとやり取りするには、マウスまたは同等のデバイスによるタッチ、およびデバイスの画面上のキーボードを使用します。Android デバイスの場合は、Android デバイスの [ホーム] または [戻る] ボタンと同じように機能するボタンが [表示コントロール] にあります。iOS デバイスの場合は、iOS デバイスのホームボタンと同じように機能する [ホーム] ボタンがあります。[最近のアプリケーション] を選択して、デバイスで実行されているアプリケーションを切り替えることもできます。

縦向きモードと横向きモードの切り替え

使用しているデバイスの縦向きと横向きを切り替えることもできます。

次のステップ

Device Farm は、手動で停止するかまたは 150 分間の時間制限に達するまでセッションを続行します。セッションを終了するには、[セッションを停止] を選択します。セッションが停止すると、キャプチャされたビデオと生成されたログにアクセスできます。詳細については、「セッションの結果を取得する」を参照してください。

ヒントとコツ

一部の AWS リージョンでは、リモートアクセスセッションでパフォーマンス問題が発生する場合があります。この原因のひとつは、一部のリージョンにおけるレイテンシーです。パフォーマンス問題が発生した場合は、リモートアクセスセッションが追いついてから、次のアプリケーション操作を行うようにします。

AWS Device Farm でリモートアクセスセッションの結果を取得する

セッションの詳細に関しては、「セッション」を参照してください。

- 前提条件

- セッション詳細の表示
- セッションのビデオやログのダウンロード

前提条件

- セッションを完了します。「AWS Device Farm でリモートアクセスセッションを使用する」の指示に従って、このページに戻ります。

セッション詳細の表示

リモートアクセスセッションが終了すると、Device Farm コンソールには、セッション中のアクティビティの詳細を含むテーブルが表示されます。詳細については、「ログ情報の分析」を参照してください。

後でセッションの詳細に戻るには:

1. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
2. セッションを含むプロジェクトを選択します。
3. [リモートアクセス] を選択し、点検するセッションをリストから選択します。

セッションのビデオやログのダウンロード

リモートアクセスセッションが終了すると、Device Farm コンソールでは、セッションのビデオキャプチャとアクティビティログにアクセスできます。セッション結果で、セッションのビデオとログへのリンクのリストを表示するには、[ファイル] タブを選択します。これらのファイルは、ブラウザで表示したり、ローカルで保存したりできます。

AWS Device Farm でのプライベートデバイスによる作業

プライベートデバイスとは、Amazon データセンターで AWS Device Farm がユーザーに代わってデプロイする物理モバイルデバイスです。このデバイスはお客様のアカウント専用です AWS 。

> **Note**
>
> 現在、プライベートデバイスは AWS 米国西部 (オレゴン) リージョン () でのみ使用できますus-west-2。

プライベートデバイスのフリートがある場合は、リモートアクセスセッションを作成し、プライベートデバイスでテストランをスケジュールすることができます。また、インスタンスプロファイルを作成して、リモートアクセスセッションまたはテストランのプライベートデバイスの動作を制御することもできます。詳細については、「AWS Device Farm ファームでのプライベートデバイスの管理」を参照してください。オプションで、特定の Android プライベートデバイスをルートデバイスとしてデプロイするようにリクエストできます。

また、Amazon Virtual Private Cloud エンドポイントサービスを作成し、会社がアクセスできるプライベートアプリケーションをテストすることもできますが、これらのアプリはインターネット経由で到達することができません。例えば、モバイルデバイスでテストする、VPC 内で実行中のウェブアプリケーションなどです。詳細については、「Device Farm での Amazon VPC エンドポイントサービスの使用 - レガシー (非推奨)」を参照してください。

複数プライベートデバイスのフリートの使用にご関心をお持ちの場合は、お問い合わせください。Device Farm チームは、お客様と協力して、 AWS アカウントのプライベートデバイスのフリートをセットアップおよびデプロイする必要があります。

トピック

- AWS Device Farm ファームでのプライベートデバイスの管理
- デバイスプール内のプライベートデバイスの選択
- AWS Device Farm でのプライベートデバイスにおけるアプリケーション再署名のスキップ
- AWS リージョン間での Amazon VPC による作業
- プライベートデバイスの終了

AWS Device Farm ファームでのプライベートデバイスの管理

プライベートデバイスとは、Amazon データセンターで AWS Device Farm がユーザーに代わってデプロイする物理モバイルデバイスです。このデバイスは、お使いの AWS アカウント専用です。

> **Note**
>
> 現在、プライベートデバイスは、AWS 米国西部 (オレゴン) リージョン (us-west-2) でのみ使用できます。

1 つ以上のプライベートデバイスを含むフリートをセットアップできます。これらのデバイスは、お客様の AWS アカウント専用です。デバイスをセットアップしたら、それらのデバイスの 1 つ以上のインスタンスプロファイルをオプションで作成できます。インスタンスプロファイルは、テスト実行を自動化し、同じ設定をデバイスインスタンスに一貫して適用するのに役立ちます。

このトピックでは、インスタンスプロファイルを作成し、他の一般的なデバイス管理タスクを実行する方法について説明します。

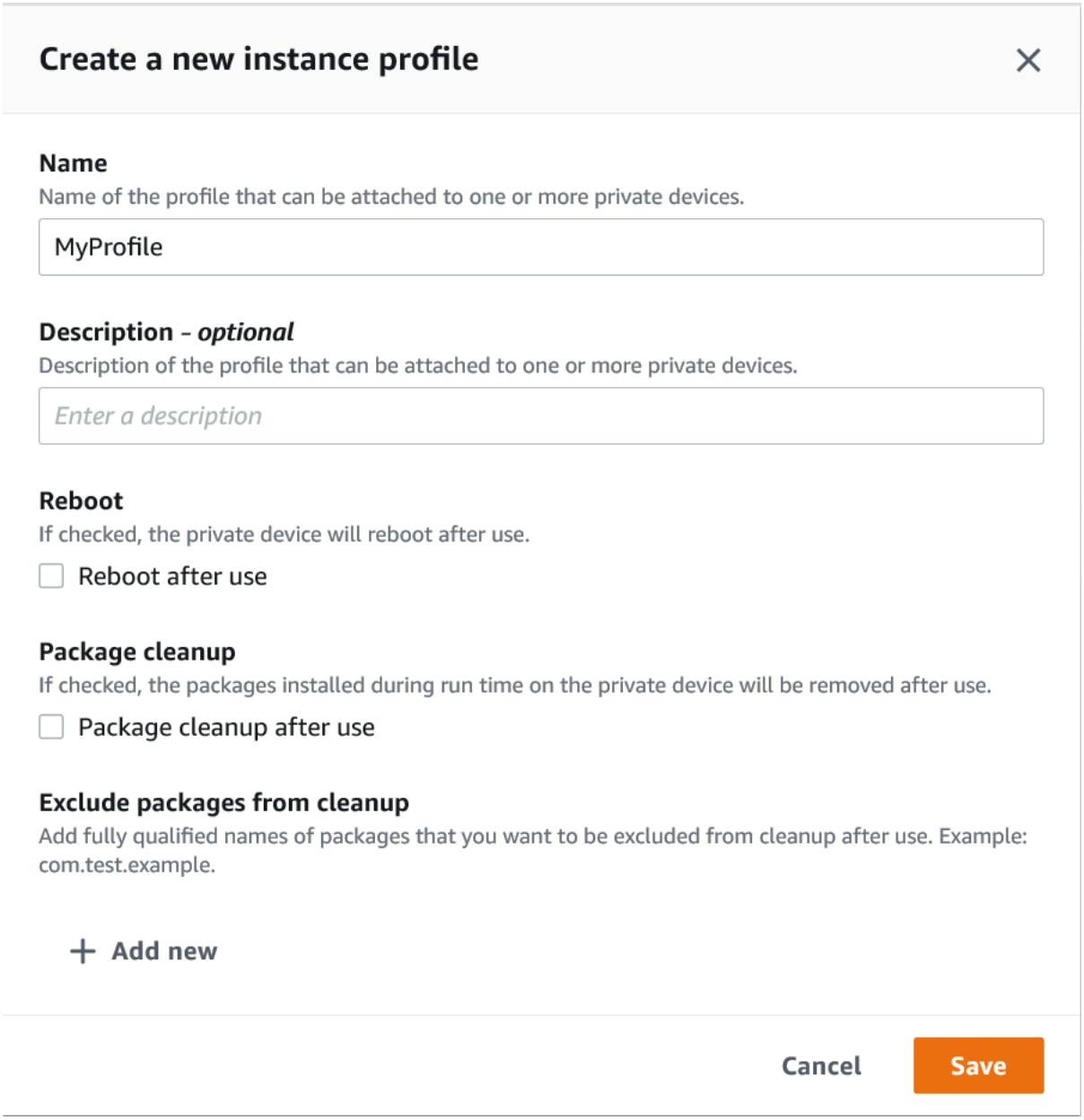
トピック

インスタンスプロファイルの作成

テスト実行中またはリモートアクセスセッション中にプライベートデバイスの動作を制御するには、Device Farm のインスタンスプロファイルを作成または変更します。プライベートデバイスの使用を開始するために、インスタンスプロファイルは必要ありません。

1. https://console.aws.amazon.com/devicefarm/ で Device Farm コンソールを開きます。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択後、[プライベートデバイス] を選択します。
3. [インスタンスプロファイル] を選択します。
4. [インスタンスプロファイルを作成] を選択します。

5. インスタンスプロファイルの名前を入力します。



6. (オプション) インスタンスプロファイルの説明を入力します。
7. (オプション) 各テスト実行またはセッションの終了後に Device Farm がデバイスに対して実行するアクションを指定するには、次の設定を変更します:

   - [使用後に再起動] - デバイスを再起動するには、このチェックボックスをオンにします。デフォルトでは、このチェックボックスはオフ (`false`) になっています。
   - [パッケージのクリーンアップ] - デバイスにインストール済みのアプリケーションパッケージをすべて削除するには、このチェックボックスをオンにします。デフォルトでは、このチェッ

クボックスはオフ (false) になっています。デバイスにインストール済みのアプリケーションパッケージをすべて保持するには、このチェックボックスをオフにします。

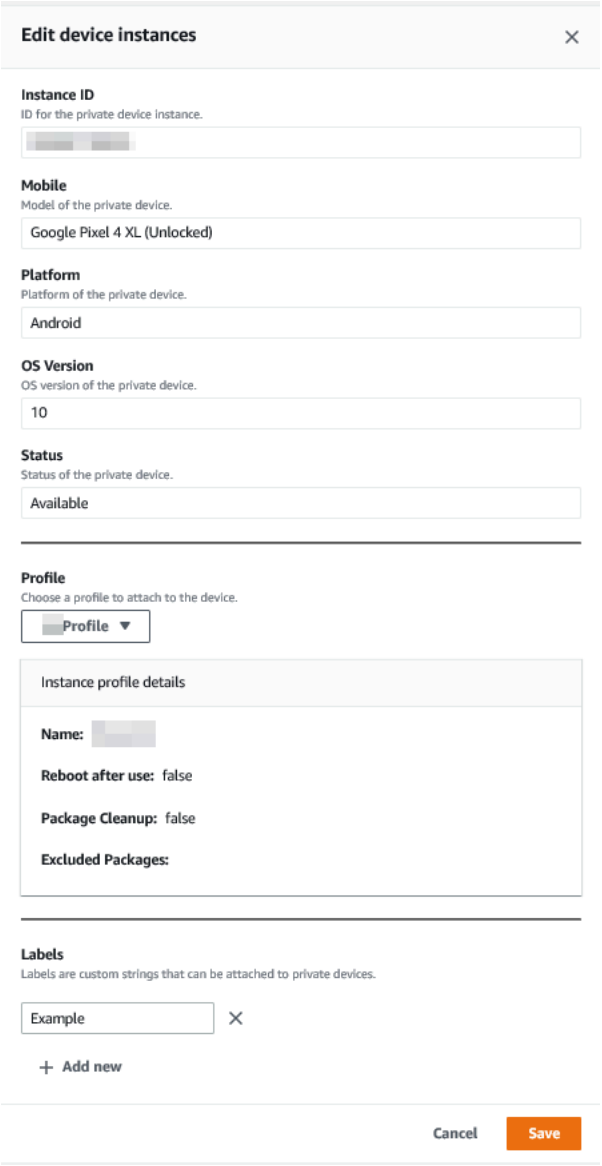
- [クリーンアップからパッケージを除外] - 選択したアプリケーションパッケージのみをデバイスに保存するには、[パッケージのクリーンアップ] チェックボックスをオンにして、[新規追加 をオンにします。パッケージ名に、デバイスで維持するアプリケーションパッケージの完全修飾名 (例: com.test.example) を入力します。さらに多くのアプリケーションパッケージをデバイスに保存するには、[新規追加] を選択して、各パッケージの完全修飾名を入力します。

8. [保存 を選択します。

プライベートデバイスインスタンスの管理

フリートにすでに 1 つ以上のプライベートデバイスがある場合は、各デバイスインスタンスに関する情報を表示したり、特定の設定を管理したりできます。追加のプライベートデバイスインスタンスをリクエストすることもできます。

1. https://console.aws.amazon.com/devicefarm/ で Device Farm コンソールを開きます。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択後、[プライベートデバイス] を選択します。
3. [デバイスインスタンス] を選択します。[デバイスインスタンス] タブには、フリートにあるプライベートデバイスのテーブルが表示されます。テーブルをすばやく検索またはフィルタリングするには、列の上の検索バーに検索語を入力します。
4. (オプション) 新しいプライベートデバイスインスタンスをリクエストするには、[デバイスインスタンスをリクエスト] を選択するか、[お問い合わせ] を選択します。プライベートデバイスでは、Device Farm チームのサポートを受けながら、追加セットアップを行う必要があります。
5. デバイスインスタンスのテーブルで、情報を表示または管理するインスタンスの横にあるトグルオプションを選択して、[編集] を選択します。



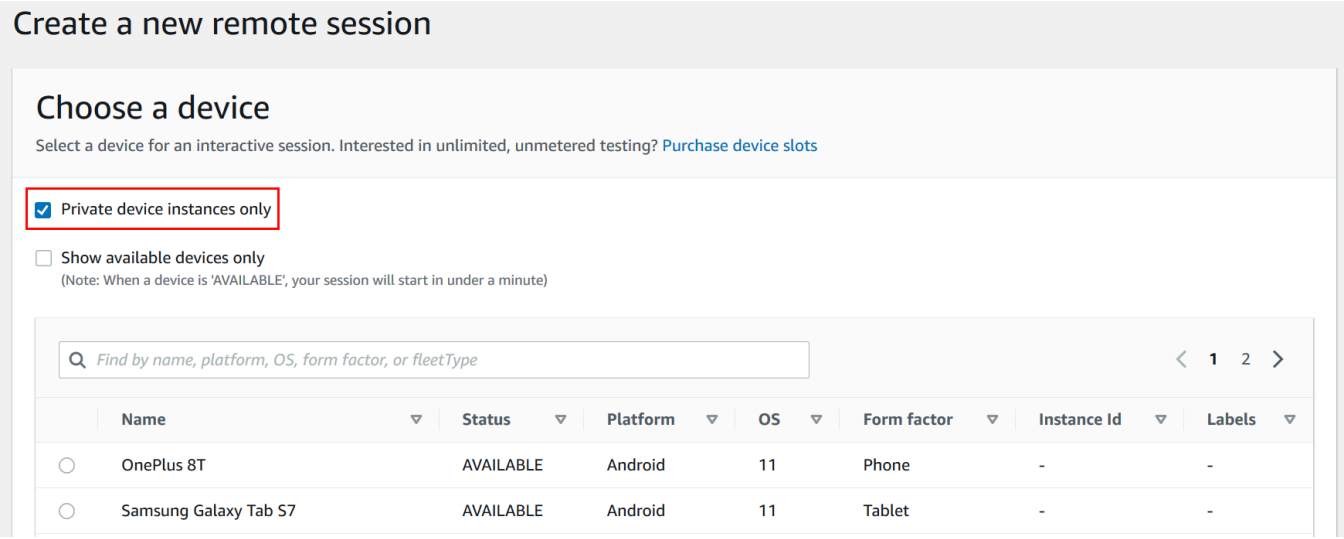
6. (オプション) [プロファイル] で、デバイスインスタンスにアタッチするインスタンスプロファイルを選択します。例えば、特定のアプリケーションパッケージを常にクリーンアップタスクから除外する場合に便利です。
7. (オプション) [ラベル] で、[新規追加] を選択して、ラベルをデバイスインスタンスに追加します。ラベルを使用すると、デバイスを分類して、特定のデバイスを簡単に見つけることができます。
8. [保存] を選択します。

テスト実行の作成またはリモートアクセスセッションの開始

プライベートデバイスフリートをセットアップしたら、テスト実行を作成したり、フリート内の 1 つ以上のプライベートデバイスでリモートアクセスセッションを開始したりできます。

1. https://console.aws.amazon.com/devicefarm/ で Device Farm コンソールを開きます。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. リストから既存のプロジェクトを選択するか、新規プロジェクトを作成します。新規プロジェクトを作成する場合は、[新規プロジェクト] を選択し、プロジェクトの名前を入力し、[送信] を選択します。
4. 次のいずれかを実行します:

   - テスト実行を作成するには、[自動テスト] を選択後、[新規実行を作成] を選択します ウィザードで、実行を作成するステップを進みます。[デバイスを選択] ステップでは、既存のデバイスプールを編集するか、Device Farm チームがセットアップし AWS アカウントに関連付けられているプライベートデバイスのみを含む、新規デバイスプールを作成します。詳細については、「the section called “プライベートデバイスプールを作成する”」を参照してください。
   - リモートアクセスセッションを開始するには、[リモートアクセス] を選択し、[新規セッションを開始] を選択します。「デバイスを選択する」ページで、[プライベートデバイスインスタンスのみ] を選択して、Device Farm チームがセットアップし、AWS アカウントに関連付けられているプライベートデバイスのみ含まれるようにリストを制限します。次に、アクセスするデバイスを選択し、リモートアクセスセッションの名前を入力して、[セッションを確定して開始] を選択します。



次のステップ

プライベートデバイスをセットアップしたら、次の方法でプライベートデバイスを管理することもできます:

- プライベートデバイスでアプリケーション再署名をスキップする
- Device Farm で Amazon Virtual Private Cloud エンドポイントサービスを使用する

インスタンスプロファイルを削除するには、[インスタンスプロファイル] メニューで、削除するインスタンスの横にあるトグルオプションを選択してから、[削除] を選択します。

デバイスプール内のプライベートデバイスの選択

テスト実行でプライベートデバイスを使用するために、プライベートデバイスを選択するデバイスプールを作成できます。デバイスプールでは、主に 3 種類のデバイスプールルールを通してプライベートデバイスを選択できます。

1. デバイス ARN に基づくルール
2. デバイスインスタンスラベルに基づくルール
3. デバイスインスタンス ARN に基づくルール

以下のセクションでは、各ルールタイプとユースケースについて詳しく説明します。Device Farm コンソール、AWS コマンドラインインターフェイス (AWS CLI)、または Device Farm API を使用し

て、これらのルールを使用してプライベートデバイスを含むデバイスプールを作成または変更できます。

トピック

デバイス ARN

デバイス ARN は、特定の物理デバイスインスタンスではなく、デバイスのタイプを表す識別子です。デバイスタイプは次の属性によって定義されます:

- デバイスのフリート ID
- デバイスの OEM
- デバイスのモデル番号
- デバイスのオペレーティングシステムのバージョン
- ルート化されているかどうかを示すデバイス状態

多くの物理デバイスインスタンスは 1 つのデバイスタイプで表すことができ、そのタイプのすべてのインスタンスには同じ属性値が割り当てられます。たとえば、プライベートフリートに iOS バージョン *16.1.0* を搭載した *Apple iPhone 13* デバイスが 3 台ある場合、各デバイスは同じデバイス ARN を共有することになります。同じ属性を持つデバイスがフリートに追加または削除された場合でも、デバイス ARN はそのデバイスタイプでフリートで使用可能なデバイスを表し続けます。

デバイス ARN は、デバイスプールのプライベートデバイスを選択する最も堅牢な方法です。デバイス ARN を使用すると、任意の時点でデプロイした特定のデバイスインスタンスに関係なく、デバイスプールがデバイスを選択し続けることができるからです。個々のプライベートデバイスインスタンスにはハードウェア障害が発生する可能性があり、Device Farm はそれらを同じデバイスタイプの新しい動作インスタンスに自動的に交換するよう求められます。このようなシナリオでは、デバイス ARN ルールにより、ハードウェア障害が発生した場合でもデバイスプールが引き続きデバイスを選択できるようになります。

デバイスプール内のプライベートデバイスにデバイス ARN ルールを使用し、そのプールでのテスト実行をスケジュールすると、Device Farm はどのプライベートデバイスインスタンスがそのデバイス ARN で表されているかを自動的に確認します。現在利用可能なインスタンスのうち、そのうちの 1 つがテストの実行に割り当てられます。現在利用可能なインスタンスがない場合、Device Farm はそのデバイス ARN の最初の利用可能なインスタンスが使用可能になるのを待ち、それを割り当ててテストを実行します。

デバイスインスタンスラベル

デバイスインスタンスラベルは、デバイスインスタンスのメタデータとしてアタッチできるテキスト識別子です。各デバイスインスタンスには複数のラベルを、複数のデバイスインスタンスには同じラベルをアタッチできます。デバイスインスタンスでデバイスラベルを追加、変更、削除する方法について詳しくは、「プライベートデバイスの管理」を参照してください。

デバイスインスタンスラベルは、デバイスプール用のプライベートデバイスを確実に選択する方法になります。同じラベルのデバイスインスタンスが複数ある場合、デバイスプールはそれらの中からテスト対象として 1 つを選択できるからです。デバイス ARN がユースケースに適さない場合 (たとえば、複数のデバイスタイプのデバイスから選択する場合や、デバイスタイプのすべてのデバイスのサブセットから選択する場合)、デバイスインスタンスラベルを使用すると、デバイスプールの複数デバイスをより細かく選択できます。個々のプライベートデバイスインスタンスにはハードウェア障害が発生する可能性があり、Device Farm はそれらを同じデバイスタイプの新しい動作インスタンスに自動的に交換するよう求められます。このようなシナリオでは、交換後のデバイスインスタンスには、交換されたデバイスのインスタンスラベルメタデータは保持されません。そのため、同じデバイスインスタンスラベルを複数のデバイスインスタンスに適用すると、デバイスインスタンスラベルルールにより、ハードウェア障害が発生した場合でもデバイスプールが引き続きデバイスインスタンスを選択できるようになります。

デバイスプール内のプライベートデバイスにデバイスインスタンスラベルルールを使用し、そのプールでのテスト実行をスケジュールすると、Device Farm はどのプライベートデバイスインスタンスがそのデバイスインスタンスラベルで表されているかを自動的に確認し、それらのインスタンスのうち、テストを実行できるインスタンスをランダムに選択します。使用可能なものがない場合、Device Farm はデバイスインスタンスラベルの付いたデバイスインスタンスをランダムに選択してテストを実行し、使用可能になったらデバイス上で実行するテストをキューに入れます。

インスタンス ARN

デバイスインスタンス ARN は、プライベートフリートにデプロイされた物理ベアメタルデバイスインスタンスを表す識別子です。たとえば、プライベートフリートの OS *15.0.0* で 3 台の *iPhone*

13 デバイスがある場合、各デバイスは同じデバイス ARN を共有しますが、各デバイスはそのインスタンスだけを表す独自のインスタンス ARN も持ちます。

デバイスインスタンス ARN は、デバイスプール用のプライベートデバイスを選択する最も堅牢でない方法であり、デバイス ARN とデバイスインスタンスラベルがユースケースに合わない場合にのみ推奨されます。デバイスインスタンス ARN は、テストの前提条件として特定のデバイスインスタンスを独自の方法で構成する場合や、テストを実行する前にその構成を確認して検証する必要がある場合に、デバイスプールのルールとしてよく使用されます。個々のプライベートデバイスインスタンスにはハードウェア障害が発生する可能性があり、Device Farm はそれらを同じデバイスタイプの新しい動作インスタンスに自動的に交換するよう求められます。これらのシナリオでは、交換用デバイスインスタンスのデバイスインスタンス ARN は、交換されたデバイスとは異なります。そのため、デバイスプールにデバイスインスタンス ARN を使用している場合は、デバイスプールのルール定義を古い ARN の使用から新しい ARN の使用に手動で変更する必要があります。テスト用にデバイスを手動で事前構成する必要がある場合、これは (デバイス ARN と比較して) 効果的なワークフローになる可能性があります。大規模なテストを行う場合は、これらのユースケースをデバイスインスタンスラベルに合わせて調整し、可能であれば複数のデバイスインスタンスをテスト用に事前構成しておくことをお勧めします。

デバイスプール内のプライベートデバイスにデバイスインスタンス ARN ルールを使用し、そのプールでのテスト実行をスケジュールすると、Device Farm はそのテストをそのデバイスインスタンスに自動的に割り当てます。そのデバイスインスタンスが使用できない場合、使用可能になると Device Farm がデバイスでテストをキューに入れます。

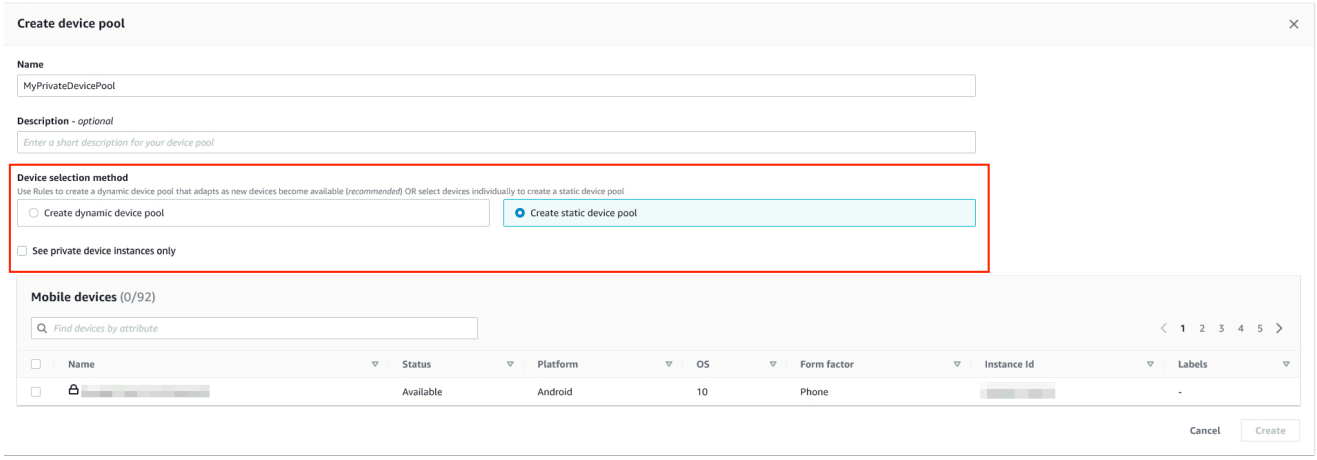
プライベートデバイスによるプライベートデバイスプールの作成 (コンソール)

テスト実行を作成するときに、テスト実行用のデバイスプールを作成し、そのプールにプライベートデバイスのみが含まれるようにします。

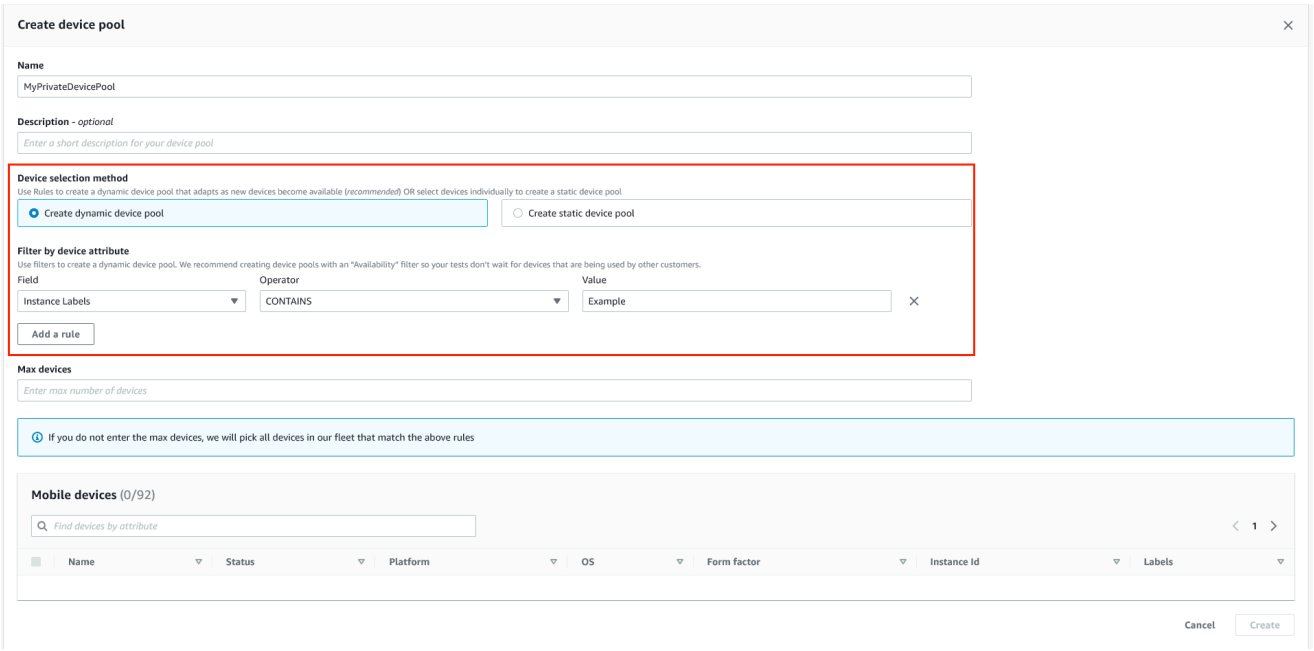
> **Note**
>
> コンソールでプライベートデバイスを使用してデバイスプールを作成する場合、プライベートデバイスの選択に使用できるのは 3 つのルールの 1 つだけです。プライベートデバイス用の複数のタイプのルールを含むデバイスプール (たとえば、デバイス ARN とデバイスインスタンス ARN のルールを含むデバイスプール) を作成する場合は、CLI または API を使用してプールを作成する必要があります。

1. https://console.aws.amazon.com/devicefarm/ で Device Farm コンソールを開きます。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. リストから既存のプロジェクトを選択するか、新しいプロジェクトを作成します。新しいプロジェクトを作成する場合は、[新規プロジェクト] を選択し、プロジェクトの名前を入力し、[送信] を選択します。
4. [自動テスト] を選択後、[新規実行を作成] を選択します。ウィザードの指示に従ってアプリケーションを選択し、実行するテストを構成します。
5. [デバイスを選択] ステップで、[デバイスプールを作成] を選択し、デバイスプールの名前とオプションの説明を入力します。

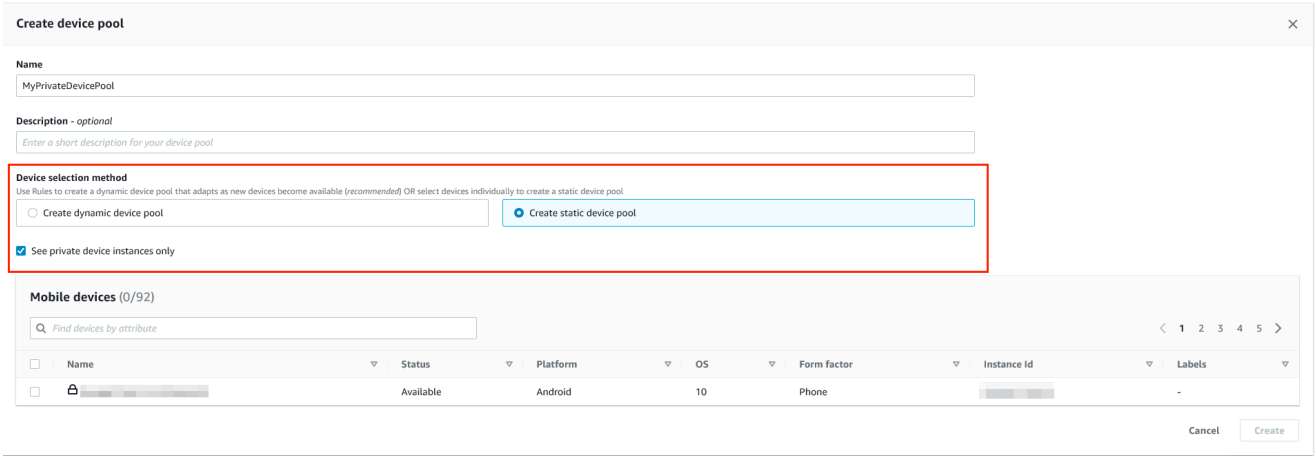
   a. デバイスプールでデバイス ARN ルールを使用するには、[静的デバイスプールを作成] を選択し、デバイスプールで使用したいリストから特定のデバイスタイプを選択します。プライベートデバイスインスタンスだけを選択しないでください。このオプションでは、デバイスプールが (デバイス ARN ルールの代わりに) デバイスインスタンス ARN ルールを使用して作成されるためです。



   b. デバイスプールにデバイスインスタンスラベルルールを使用するには、[動的デバイスプールを作成] を選択します。次に、デバイスプールで使用したいラベルごとに [ルールを追加] を選択します。ルールごとに、[インスタンスラベル] を `Field` として選択し、[含む] を `Operator` として選択し、必要なデバイスインスタンスラベルを `Value` として指定します。



c. デバイスプールにデバイスインスタンス ARN ルールを使用するには、[静的デバイスプールを作成] を選択し、[プライベートデバイスインスタンスのみ] を選択して、Device Farm が AWS アカウントに関連付けたプライベートデバイスインスタンスのみにデバイスのリストを限定します。



6. [作成] を選択します。

プライベートデバイスを含むプライベートデバイスプールの作成 (AWS CLI)

- create-device-pool コマンドを実行します。

AWS CLI での Device Farm の使用についての詳細は、「AWS CLI リファレンス」を参照してください。

プライベートデバイスによるプライベートデバイスプールの作成 (API)

- `CreateDevicePool` API を呼び出します。

Device Farm API の使用についての詳細は、「Device Farm の自動化」を参照してください。

AWS Device Farm でのプライベートデバイスにおけるアプリケーション再署名のスキップ

アプリケーション署名は、デバイスにインストールしたり、Google Play Store や Apple App Store などのアプリストアに公開したりする前に、プライベートキーを使用してアプリケーションパッケージ (APK、IPA など) にデジタル署名するプロセスです。必要な署名とプロファイルの数を減らしてテストを効率化し、リモートデバイスのデータセキュリティを向上させるために、AWS Device Farm はサービスにアップロードされた後にアプリケーションに再署名します。

アプリケーションを AWS Device Farm にアップロードすると、サービスは独自の署名証明書とプロビジョニングプロファイルを使用して、アプリの新しい署名を生成します。このプロセスは、元のアプリケーション署名を AWS Device Farm の署名に置き換えます。その後、AWS Device Farm が提供するテストデバイスに再署名されたアプリケーションがインストールされます。新しい署名により、元のデベロッパーの証明書を必要とせずに、これらのデバイスにアプリケーションをインストールして実行できます。

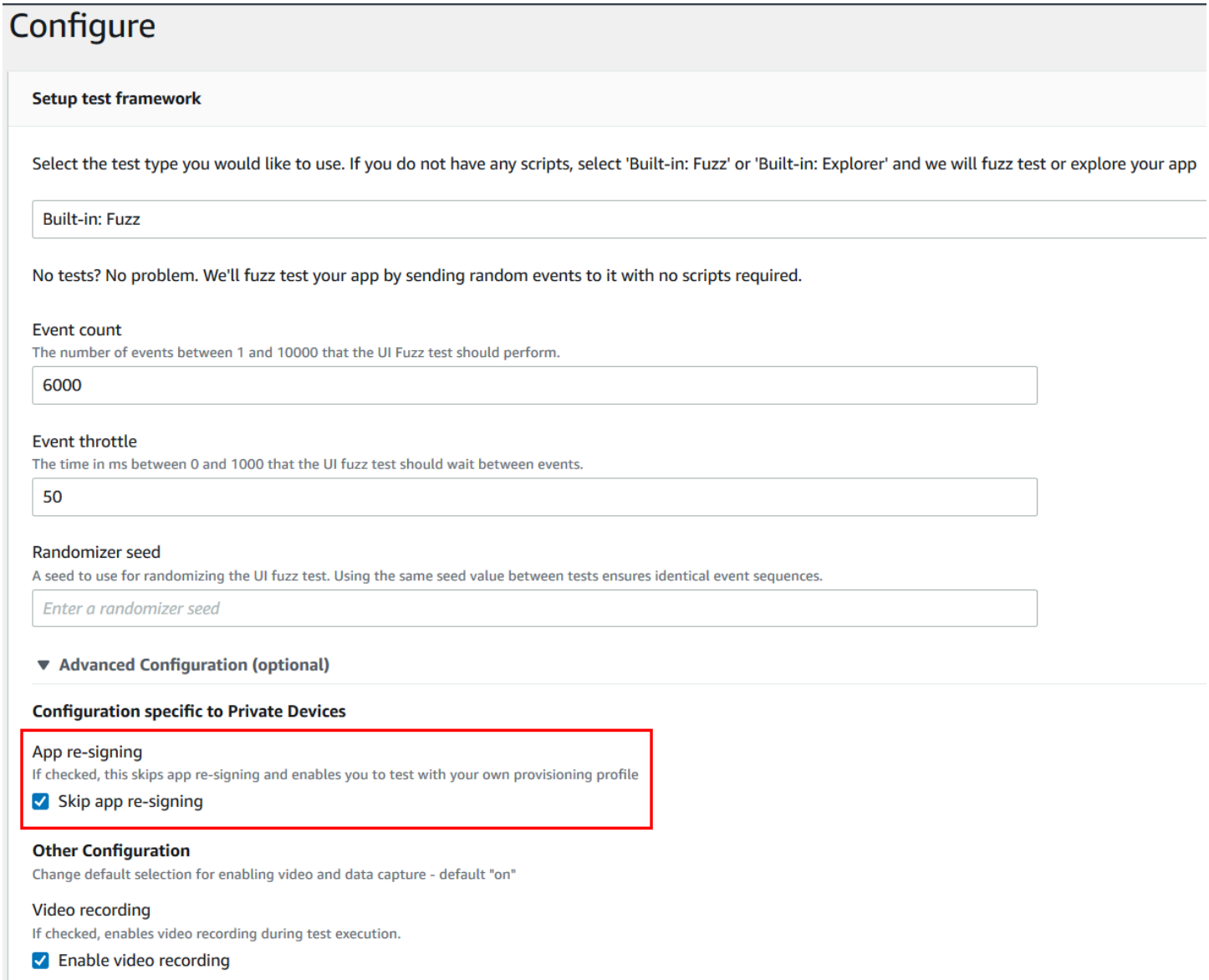
iOS では、埋め込みプロビジョニングプロファイルをワイルドカードプロファイルに置き換え、アプリケーションに再署名します。これを指定すると、インストール前に補助データがアプリケーションパッケージに追加され、データがアプリケーションのサンドボックスに表示されます。iOS アプリを辞退すると、特定の使用権限が削除されます。これには、アプリグループ、関連ドメイン、ゲームセンター HealthKit HomeKit、ワイヤレスアクセサリ設定、アプリ内購入、アプリ間オーディオ、Apple Pay、プッシュ通知、VPN 設定とコントロールが含まれます。

Android では、アプリケーションに署名します。これにより、Google Maps Android API など、アプリの署名に依存する機能が破損する可能性があります。また、 などの製品から利用できる著作権侵害防止や改ざん防止の検出をトリガーすることもあります DexGuard。組み込みテストでは、スクリーンショットのキャプチャと保存に必要なアクセス許可を含めるようにマニフェストを変更する場合があります。

プライベートデバイスを使用する場合は、AWS Device Farm がアプリケーションに再署名するステップをスキップできます。これは、Device Farm が Android および iOS プラットフォームのアプリケーションに常に再署名するパブリックデバイスとは異なります。

リモートアクセスセッションまたはテスト実行を作成する場合は、アプリケーションの再署名をスキップできます。これは、Device Farm がアプリケーションに再署名すると中断する機能がアプリケーションにある場合に役立ちます。例えば、プッシュ通知は再署名後に動作しない場合があります。Device Farm がアプリケーションをテストするときに行う変更の詳細については、AWS Device Farm FAQsまたは Apps ページを参照してください。

テスト実行でアプリケーション再署名をスキップするには、テスト実行を作成するときの「構成する」ページで、[アプリケーション再署名をスキップ] を選びます。



> **Note**
>
> XCTest フレームワークを使用している場合、[アプリケーション再署名をスキップ] オプションは選択できません。詳細については、「iOS 用 XCTest と AWS Device Farm による作業」を参照してください。

アプリケーション署名設定を構成するための追加手順は、プライベートデバイスが Android か iOS によって異なります。

Android デバイスでのアプリケーション再署名のスキップ

Android のプライベートデバイスでアプリケーションをテストする場合は、テスト実行またはリモートアクセスセッションを作成するときに、[アプリケーション再署名をスキップ] を選択します。必要な構成は他にありません。

iOS デバイスでのアプリケーション再署名のスキップ

Apple では、デバイスにロードする前に、アプリケーションにテスト用の署名を行う必要があります。iOS デバイスでは、アプリケーションの署名には 2 つのオプションがあります。

- 社内 (エンタープライズ) 開発者プロファイルを使用している場合は、次のセクション (the section called “アプリケーションを信頼するためにリモートアクセスセッションを作成する”) に進んでください。

- アドホックの iOS アプリケーション開発プロファイルを使用している場合は、まず Apple 開発者アカウントを使用してデバイスを登録してから、プロビジョニングファイルを更新してプライベートデバイスを含めます。次に、更新したプロビジョニングプロファイルでアプリケーションに再署名する必要があります。その後、Device Farm で再署名したアプリケーションを実行できます。

アドホック iOS アプリケーション開発プロビジョニングプロファイルでデバイスを登録するには

1. Apple 開発者アカウントにサインインします。
2. コンソールの [証明書、ID およびプロファイル] セクションに移動します。
3. [デバイス] に移動します。
4. デバイスを Apple 開発者アカウントで登録します。デバイスの名前と UDID を取得するには、Device Farm API の `ListDeviceInstances` オペレーションを使用します。

5. プロビジョニングプロファイルに移動して、[編集] を選択します。
6. リストからデバイスを選択します。
7. Xcode で、更新されたプロビジョニングプロファイルを取得し、アプリケーションに再署名します。

必要な構成は他にありません。これで、リモートアクセスセッションまたはテスト実行を作成し、[アプリケーション再署名をスキップ] を選択できます。

iOS アプリケーションを信頼するためのリモートアクセスセッションの作成

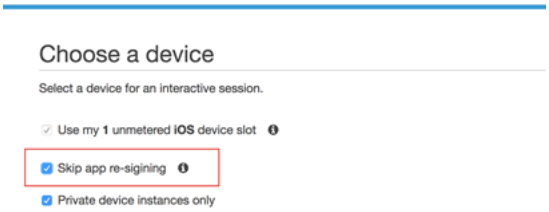
社内 (エンタープライズ) 開発者プロビジョニングプロファイルを使用している場合、1 回限りの手順を実行して、各プライベートデバイスで社内アプリケーション開発者の証明書を信頼する必要があります。

これを行うには、テストするアプリをプライベートデバイスにインストールするか、テストするアプリと同じ証明書で署名された「ダミー」アプリケーションをインストールできます。同じ証明書で署名されたダミーアプリケーションのインストールには利点があります。構成プロファイルやエンタープライズアプリケーション開発者を信頼すると、その開発者のすべてのアプリケーションは、削除するまでプライベートデバイス上で信頼されることになります。このため、テストする新しいバージョンのアプリケーションをアップロードするときに、再びアプリケーション開発者を信頼する手続きは必要ありません。これは、テスト自動化を実行し、アプリケーションのテストのたびにリモートアクセスセッションを作成したくない場合に特に便利です。

リモートアクセスセッションを開始する前には、「インスタンスプロファイルの作成」の手順に従って、Device Farm でインスタンスプロファイルを作成または変更します。インスタンスプロファイルで、テストアプリケーションまたはダミーアプリケーションのバンドル ID を [クリーンアップからパッケージを除外] 設定に追加します。その後、このインスタンスプロファイルをプライベートデバイスインスタンスにアタッチして、新しいテスト実行の開始前に Device Farm がデバイスからこのアプリケーションを削除しないようにします。これにより、開発者の証明書は信頼されたままとなります。

ダミーアプリケーションはリモートアクセスセッションを使用してデバイスにアップロードでき、アプリケーションを起動し、開発者を信頼できます。

1. 「セッションを作成する」の手順に従って、作成したプライベートデバイスインスタンスプロファイルを使用するリモートアクセスセッションを作成します。セッションを作成するときは、必ず [アプリケーション再署名をスキップ] を選択します。



> **Important**
>
> プライベートデバイスのみを含むようにデバイスのリストをフィルタリングするには、必ず正しいインスタンスプロファイルでプライベートデバイスを使用するように [プライベートデバイスインスタンスのみ] を選択します。

ダミーアプリケーションまたはテストするアプリケーションも、このインスタンスにアタッチされているインスタンスプロファイルの [クリーンアップからパッケージを除外] 設定に必ず追加してください。

2. リモートセッションが開始したら、[ファイルを選択] を選択し、社内プロビジョニングプロファイルを使用するアプリケーションをインストールします。
3. アップロードしたアプリケーションを起動します。
4. 開発者証明書を信頼する手順に従います。

この構成プロファイルまたはエンタープライズアプリケーション開発者のすべてのアプリケーションは、削除するまでこのプライベートデバイス上で信頼されるようになりました。

AWS リージョン間での Amazon VPC による作業

Device Farm サービスは米国西部 (オレゴン) (`us-west-2`) リージョンに限られます。Amazon Virtual Private Cloud (Amazon VPC) を使用すると、Device Farm を使用して別の AWS リージョンの Amazon Virtual Private Cloud のサービスを使用できます。Device Farm とお使いのサービスが同じリージョンの場合は、「Device Farm での Amazon VPC エンドポイントサービスの使用 - レガシー (非推奨)」を参照してください。

別リージョンのプライベートサービスにアクセスするには、次の 2 つの方法があります。`us-west-2` ではない別リージョンのサービスを使用する場合は、VPC ピアリングを使用して、そのリージョンの VPC を `us-west-2` の Device Farm とインターフェイス接続している別の VPC とピ

アリングできます。ただし、複数リージョンのサービスを使用する場合、Transit Gateway を使えば、より簡単なネットワーク構成でそれらのサービスにアクセスできます。

詳細については、「Amazon VPC ピアリングガイド」の「VPC ピアリングのシナリオ」を参照してください。

VPC ピアリング

別リージョンの 2 つの VPC をピアリングすることは、重複しない個別の CIDR ブロックがある場合に可能です。これにより、プライベート IP アドレスがすべて一意であることが保証され、VPC 内のすべてのリソースが、任意の形式のネットワークアドレス変換 (NAT) を必要とせずに相互に対応できるようになります。CIDR 表記の詳細については、「RFC 4632」を参照してください。

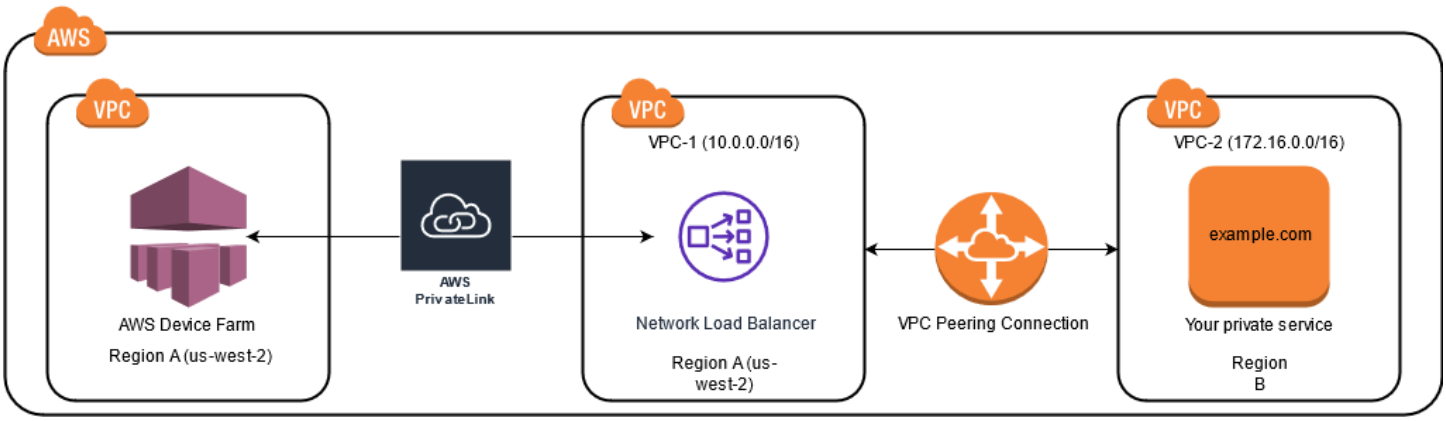
このトピックには、Device Farm (「VPC-1」と呼ばれます) が米国西部 (オレゴン) (`us-west-2`) リージョンにある、 クロスリージョンサンプルシナリオが含まれます。このサンプルの 2 番目の VPC (「VPC-2」と呼ばれます) は別リージョンにあります。

Device Farm VPC クロスリージョンサンプル

VPC コンポーネント	VPC-1	VPC-2
CIDR	10.0.0.0/16	172.16.0.0/16

> **Important**
>
> 2 つの VPC 間でピアリング接続を確立すると、VPC のセキュリティ体制が変わる可能性があります。さらに、ルートテーブルに新規エントリを追加すると、VPC 内のリソースのセキュリティ体制が変わる可能性があります。組織のセキュリティ要件を満たす方法でこれらの構成を実装するのはお客様の責任です。詳細については、「責任共有モデル」を参照してください。

次の図は、このサンプルのコンポーネントと、それらのコンポーネント間の相互作用を示しています。



トピック

- 前提条件
- ステップ 1: VPC-1 と VPC -2 間のピアリング接続を設定する
- ステップ 2: VPC-1 と VPC -2 でルートテーブルを更新する
- ステップ 3: 対象グループを作成する
- ステップ 4: Network Load Balancer を作成する
- ステップ 5: VPC エンドポイントを作成する
- ステップ 6: Device Farm での VPC エンドポイント構成を作成する
- ステップ 7: テスト実行を作成する
- Transit Gateway でスケーラブルなネットワークを作成する

前提条件

このサンプルでは、以下が必要です:

- 重複しない CIDR ブロックを含むサブネットで構成された 2 つの VPC。
- VPC -1 は us-west-2 リージョン内にあり、アベイラビリティーゾーン us-west-2a、us-west-2b、us-west-2c のサブネットを含む必要があります。

VPC の作成とサブネットの構成の詳細については、「Amazon VPC ピアリングガイド」の「VPC とサブネットによる作業」を参照してください。

ステップ 1: VPC-1 と VPC -2 間のピアリング接続を設定する

重複しない CIDR ブロックを含む 2 つの VPC 間でピアリング接続を確立します。これを行うには、「Amazon VPC ピアリングガイド」の「VPC ピアリング接続を作成および承認する」を参照してください。このトピックのクロスリージョンシナリオと「Amazon VPC ピアリングガイド」を使用すれば、次のサンプルピアリング接続構成を作成できます:

名前

```
Device-Farm-Peering-Connection-1
```

VPC ID (リクエスター)

```
vpc-0987654321gfedcba (VPC-2)
```

アカウント

```
My account
```

[リージョン]

```
US West (Oregon) (us-west-2)
```

VPC ID (アクセプター)

```
vpc-1234567890abcdefg (VPC-1)
```

> **Note**
>
> 新しいピアリング接続を確立するときは、必ず VPC ピアリング接続クォータを確認してください。詳細については、「Amazon VPC ユーザーガイド」の「Amazon VPC クォータ」を参照してください。

ステップ 2: VPC-1 と VPC -2 でルートテーブルを更新する

ピアリング接続を設定したら、2 つの VPC 間でデータを転送するための宛先ルートを確立する必要があります。このルートを確立するには、VPC-1 のルートテーブルを VPC-2 のサブネットに手動で更新できます。その逆も可能です。これを行うには、「Amazon VPC ピアリングガイド」の「VPC ピアリング接続のルートテーブルを更新する」を参照してください。このトピックのクロスリージョンシナリオと「Amazon VPC ピアリングガイド」を使用して、次のサンプルルートテーブル構成を作成します:

Device Farm VPC ルートテーブルサンプル

VPC コンポーネント	VPC-1	VPC-2
ルートテーブル ID	rtb-1234567890abcdefg	rtb-0987654321gfedcba
ローカルアドレス範囲	10.0.0.0/16	172.16.0.0/16
宛先アドレス範囲	172.16.0.0/16	10.0.0.0/16

ステップ 3: 対象グループを作成する

宛先ルートを設定したら、リクエストを VPC-2 にルーティングするように VPC -1 の Network Load Balancer を構成できます。

Network Load Balancer には、最初にリクエストの送信先の IP アドレスを含む対象グループが含まれている必要があります。

対象グループを作成するには

1. VPC-2 で対象にするサービスの IP アドレスを特定します。
   - これらの IP アドレスは、ピアリング接続で使用されるサブネットのメンバーであることが必要です。
   - 対象の IP アドレスは静的で不変でなければなりません。サービスに動的 IP アドレスがある場合は、静的リソース (Network Load Balancer など) を対象にして、その静的リソースがリクエストを実際の対象にルーティングすることを検討してください。

   > **Note**
   >
   > - 1 つ以上のスタンドアロンの Amazon Elastic Compute Cloud (Amazon EC2) インスタンスを対象にしている場合は、https://console.aws.amazon.com/ec2/ の Amazon EC2 コンソールを開き、[インスタンス] を選択します。
   > - Amazon EC2 インスタンスの Amazon EC2 Auto Scaling グループを対象にしている場合、Amazon EC2 Auto Scaling グループを Network Load Balancer に関連付ける必要があります。詳細については、「Amazon EC2 Auto Scaling ユーザーガイド」の「Auto Scaling グループへのロードバランサーのアタッチ」を参照してください。
   >
   > 次に、https://console.aws.amazon.com/ec2/ で Amazon EC2 コンソールを開き、[ネットワークインターフェイス を選択します。そこから、それぞれのアベイラ

ビリティーゾーンにある Network Load Balancer の各ネットワークインターフェースの IP アドレスを確認できます。

2. VPC-1 に対象グループを作成します。詳細については、「Network Load Balancer 用ユーザーガイド」の「Network Load Balancer の対象グループを作成する」を参照してください。

   別の VPC 内のサービスの対象グループには、次の構成が必要です:

   - 「[対象タイプを選択」で [IP アドレス] を選択します。
   - VPC の場合は、ロードバランサーをホストする VPC を選択します。トピックサンプルでは、VPC-1 になります。
   - 「対象を登録する」ページで、[VPC-2] の IP アドレスごとに対象を登録します。

     [ネットワーク] には [その他のプライベート IP アドレス] を選択します。

     [アベイラビリティーゾーン] では、[VPC-1] で目的のゾーンを選択します。

     [IPv4 アドレス] には、[VPC -2] の IP アドレスを選択します。

     [ポート] では、お使いのポートを選択します。
   - [保留中として以下を含める] をクリックします。アドレスの指定が完了したら、[保留中の対象を登録] を選択します。

このトピックのクロスリージョンシナリオと「Network Load Balancers 用ユーザーガイド」を使用すると、対象グループの構成には次の値が使用されます:

対象タイプ

```
IP addresses
```

対象グループ名

```
my-target-group
```

プロトコル/ポート

```
TCP : 80
```

VPC

```
vpc-1234567890abcdefg (VPC-1)
```

ネットワーク

```
Other private IP address
```

アベイラビリティーゾーン

```
all
```

IPv4 アドレス

```
172.16.100.60
```

ポート

```
80
```

ステップ 4: Network Load Balancer を作成する

ステップ 3 で説明した対象グループを使用してNetwork Load Balancer を作成します。これを行うには、「Network Load Balancer の作成」を参照してください。

このトピックのクロスリージョンシナリオでは、サンプルのNetwork Load Balancer 構成で次の値が使用されています:

ロードバランサー

```
my-nlb
```

スキーム

```
Internal
```

VPC

```
vpc-1234567890abcdefg (VPC-1)
```

マッピング

```
us-west-2a - subnet-4i23iuufkdiufsloi

us-west-2b - subnet-7x989pkjj78nmn23j

us-west-2c - subnet-0231ndmas12bnnsds
```

プロトコル/ポート

```
TCP : 80
```

対象グループ

```
my-target-group
```

ステップ 5: VPC エンドポイントを作成する

Network Load Balancer を使用して VPC エンドポイントサービスを作成できます。この VPC エンドポイントサービスを通じて、Device Farm は、インターネットゲートウェイ、NAT インスタンス、VPN 接続などの追加インフラストラクチャなしで VPC -2 内のサービスに接続できます。

これを行うには、「Amazon VPC エンドポイントサービスの作成」を参照してください。

ステップ 6: Device Farm での VPC エンドポイント構成を作成する

これで、VPC と Device Farm との間でプライベート接続を確立できます。Device Farm を使用し、パブリックインターネットを介して公開することなく、プライベートサービスをテストできます。これを行うには、「Device Farm での VPC エンドポイント構成の作成」を参照してください。

このトピックのクロスリージョンシナリオでは、サンプルのVPC エンドポイント構成で次の値が使用されています:

名前

```
My VPCE Configuration
```

VPCE サービス名

```
com.amazonaws.vpce.us-west-2.vpce-svc-1234567890abcdefg
```

サービス DNS 名

```
devicefarm.com
```

ステップ 7: テスト実行を作成する

ステップ 6 で説明した VPC エンドポイント構成を使用するテスト実行を作成できます。詳細については、「Device Farm でのテスト実行の作成」または「セッションを作成する」を参照してください。

Transit Gateway でスケーラブルなネットワークを作成する

3 つ以上の VPC を使用するスケーラブルなネットワークを作成するには、Transit Gateway をネットワークトランジットハブとして機能させ、VPC とオンプレミスネットワークを相互接続します。Transit Gateway を使用するように Device Farm と同じリージョンにある VPC を構成するには、「Device Farm による Amazon VPC エンドポイントサービス」ガイドに従って、プライベート IP アドレスに基づいて別のリージョンのリソースを対象にすることができます。

Transit Gateway の詳細については、「Amazon VPC Transit Gateways ガイド」の「トランジットゲートウェイについて」を参照してください。

プライベートデバイスの終了

> **Important**
>
> これらの手順は、プライベートデバイス契約の終了にのみ適用されます。その他のすべての AWS サービスおよび請求の問題については、それらの製品の各ドキュメントを参照するか、 AWS サポートにお問い合わせください。

最初の契約期間後にプライベートデバイスを終了するには、<aws-devicefarm-support@amazon.com> までメールで 30 日間の非更新通知を提出する必要があります。

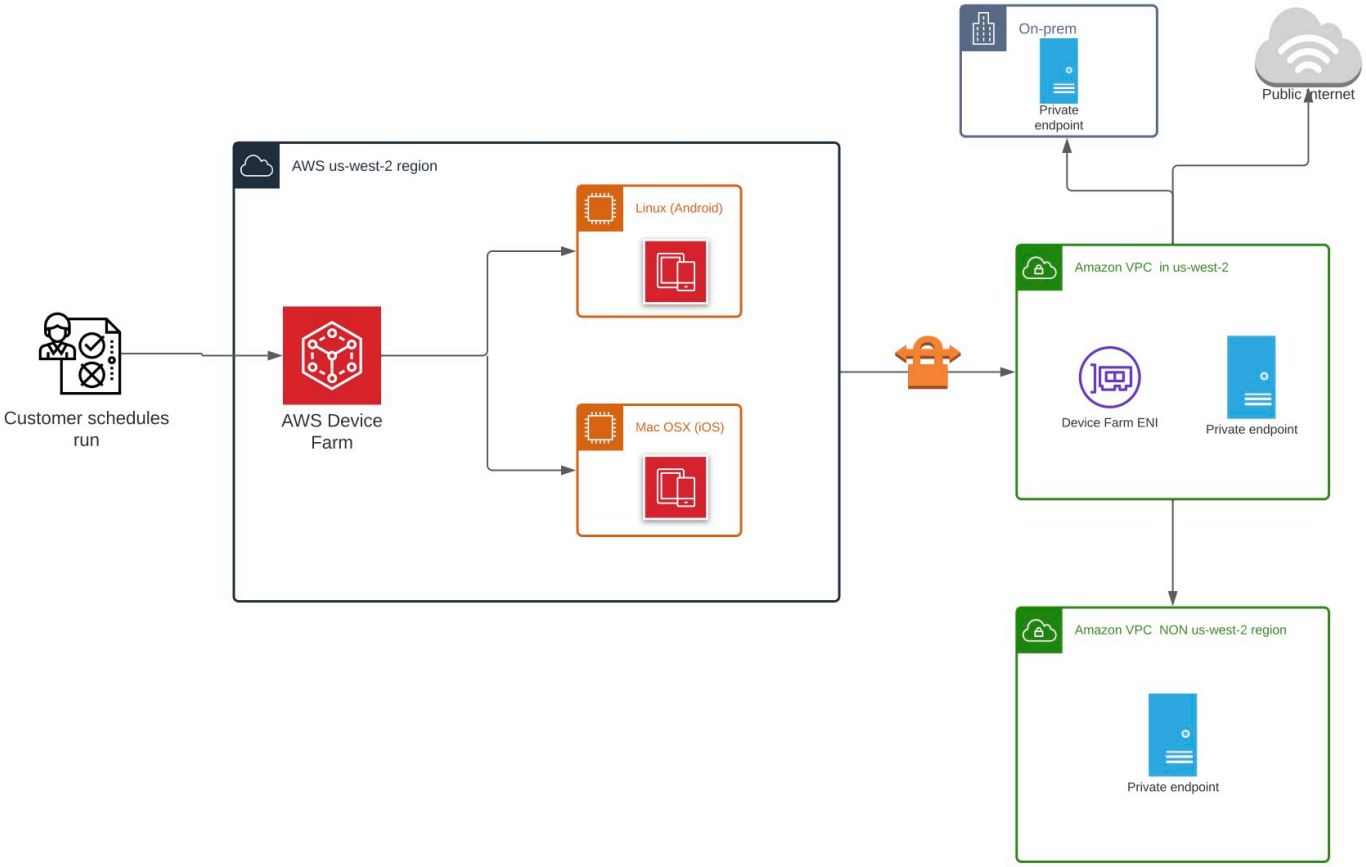
AWS Device Farm の VPC-ENI

> **Warning**
>
> この機能は、プライベートデバイスでのみ利用できます。 AWS アカウントでプライベートデバイスの使用をリクエストするには、 にお問い合わせください。 AWS アカウントにプライベートデバイスがすでに追加されている場合は、この VPC 接続方法を使用することを強くお勧めします。

AWS Device Farm の VPC-ENI 接続機能は、、、オンプレミスソフトウェア AWS、または別のクラウドプロバイダーでホストされているプライベートエンドポイントに安全に接続するのに役立ちます。

Device Farm モバイルデバイスとそのホストマシンの両方を us-west-2リージョンの Amazon Virtual Private Cloud (Amazon VPC) 環境に接続できます。これにより、Elastic Network Interface を介して分離された non-internet-facing サービスやアプリケーションにアクセスできます。VPC の詳細については、「Amazon VPC ユーザーガイド」を参照してください。

プライベートエンドポイントまたは VPC が us-west-2 リージョンにない場合は、Transit Gateway や VPC ピアリング などのソリューションを使用して us-west-2 リージョン内の VPC とリンクできます。このような場合、Device Farm は us-west-2 リージョン用に指定したサブネットに ENI を作成します。その us-west-2 リージョン VPC と他のリージョンの VPC との間で接続を確立できるようにする必要があります。



AWS CloudFormation を使用して VPCs、 のテンプレートリポジトリの VPCPeering AWS CloudFormation テンプレートを参照してください GitHub。

> **Note**
>
> Device Farm では、`us-west-2` のお客様が VPC で ENI を作成しても料金は発生しません。クロスリージョン接続または外部 VPC 間接続の費用は、この機能には含まれていません。

VPC アクセスを構成すると、VPC 内に指定した NAT ゲートウェイがない限り、テストに使用するデバイスとホストマシンは VPC 外のリソース (パブリック CDN など) に接続できなくなります。詳細については、「Amazon VPC ユーザーガイド」の「NAT ゲートウェイ」を参照してください。

トピック

AWS アクセスコントロールと IAM

AWS Device Farm では、AWS Identity and Access Management (IAM) を使用して Device Farm の機能へのアクセス権を付与または制限するポリシーを作成できます。AWS Device Farm で VPC 接続機能を使用するには、AWS Device Farm へのアクセスに使用しているユーザーアカウントまたはロールに次の IAM ポリシーが求められます:

```
{
  "Version": "2012-10-17",
  "Statement": [{
      "Effect": "Allow",
      "Action": [
        "devicefarm:*",
        "ec2:DescribeVpcs",
        "ec2:DescribeSubnets",
        "ec2:DescribeSecurityGroups",
        "ec2:CreateNetworkInterface"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iam:CreateServiceLinkedRole",
      "Resource": "arn:aws:iam::*:role/aws-service-role/devicefarm.amazonaws.com/
AWSServiceRoleForDeviceFarm",
      "Condition": {
        "StringLike": {
          "iam:AWSServiceName": "devicefarm.amazonaws.com"
        }
```

```
      }
    }
  ]
}
```

VPC 構成で Device Farm プロジェクトを作成または更新するには、IAM ポリシーで VPC 構成にリストされているリソースに対して次のアクションを呼び出すことを許可する必要があります:

```
"ec2:DescribeVpcs"
"ec2:DescribeSubnets"
"ec2:DescribeSecurityGroups"
"ec2:CreateNetworkInterface"
```

さらに、IAM ポリシーでサービスリンクロールの作成を許可する必要もあります:

```
"iam:CreateServiceLinkedRole"
```

> **Note**
>
> これらの権限はいずれも、プロジェクトで VPC 構成を使用しないユーザーには必要ありません。

サービスリンクロール

AWS Device Farm は AWS Identity and Access Management 、 (IAM) サービスにリンクされたロール を使用します。サービスリンクロールは、Device Farm に直接リンクされる一意のタイプの IAM ロールです。サービスにリンクされたロールは、Device Farm によって事前定義されており、ユーザーに代わってサービスから他の AWS のサービスを呼び出すために必要なすべてのアクセス許可が含まれています。

サービスリンクロールを使用すると、必要な権限を手動で追加する必要がないため、Device Farm のセットアップが簡単になります。Device Farm は、サービスリンクロールの権限を定義します。別の定義がなされている場合を除き、Device Farm のみがそのロールを引き受けることができます。定義される権限には、信頼ポリシーと権限ポリシーが含まれており、その権限ポリシーを他のIAM エンティティにアタッチすることはできません。

サービスリンクロールは、まずその関連リソースを削除しなければ削除できません。これにより、リソースへのアクセス権限を不用意に削除することがなくなり、Device Farm のリソースを保護できます。

サービスリンクロールをサポートする他のサービスについては、「IAM と連携する AWS サービス」で「サービスリンクロール」列が「はい」になっているサービスを探してください。サービスのサービスリンクロールに関するドキュメンテーションを表示するには、[Yes] (はい) リンクを選択します。

Device Farm のサービスリンクロール権限

Device Farm は、 という名前のサービスにリンクされたロールを使用しますAWSServiceRoleForDeviceFarm。これによりDevice Farm がユーザーに代わって AWS リソースにアクセスできるようになります。

AWSServiceRoleForDeviceFarm サービスにリンクされたロールは、次のサービスを信頼してロールを引き受けます。

- `devicefarm.amazonaws.com`

ロールの権限ポリシーは、Device Farm が次のアクションを完了することを許可します:

- アカウント用
  - ネットワークインターフェイスを作成する
  - ネットワークインターフェースを記述する
  - VPC を記述する
  - サブネットを記述する
  - セキュリティグループを記述する
  - インターフェイスを削除する
  - ネットワークインターフェイスを変更する
- ネットワークインターフェイス用
  - タグを作成する
- Device Farm によって管理される EC2 ネットワークインターフェイス用
  - ネットワークインターフェイス権限を作成する

IAM ポリシーの全文は次のとおりです:

```
{
"Version": "2012-10-17",
"Statement": [
 {
  "Effect": "Allow",
  "Action": [
   "ec2:DescribeNetworkInterfaces",
   "ec2:DescribeVpcs",
   "ec2:DescribeSubnets",
   "ec2:DescribeSecurityGroups"
  ],
  "Resource": "*"
 },
 {
  "Effect": "Allow",
  "Action": [
   "ec2:CreateNetworkInterface"
  ],
  "Resource": [
   "arn:aws:ec2:*:*:subnet/*",
   "arn:aws:ec2:*:*:security-group/*"
  ]
 },
 {
  "Effect": "Allow",
  "Action": [
   "ec2:CreateNetworkInterface"
  ],
  "Resource": [
   "arn:aws:ec2:*:*:network-interface/*"
  ],
  "Condition": {
   "StringEquals": {
    "aws:RequestTag/AWSDeviceFarmManaged": "true"
   }
  }
 },
 {
  "Effect": "Allow",
  "Action": [
   "ec2:CreateTags"
  ],
  "Resource": "arn:aws:ec2:*:*:network-interface/*",
```

```
    "Condition": {
     "StringEquals": {
      "ec2:CreateAction": "CreateNetworkInterface"
     }
    }
   },
   {
    "Effect": "Allow",
    "Action": [
     "ec2:CreateNetworkInterfacePermission",
     "ec2:DeleteNetworkInterface"
    ],
    "Resource": "arn:aws:ec2:*:*:network-interface/*",
    "Condition": {
     "StringEquals": {
      "aws:ResourceTag/AWSDeviceFarmManaged": "true"
     }
    }
   },
   {
    "Effect": "Allow",
    "Action": [
     "ec2:ModifyNetworkInterfaceAttribute"
    ],
    "Resource": [
     "arn:aws:ec2:*:*:security-group/*",
     "arn:aws:ec2:*:*:instance/*"
    ]
   },
   {
    "Effect": "Allow",
    "Action": [
     "ec2:ModifyNetworkInterfaceAttribute"
    ],
    "Resource": "arn:aws:ec2:*:*:network-interface/*",
    "Condition": {
     "StringEquals": {
      "aws:ResourceTag/AWSDeviceFarmManaged": "true"
     }
    }
   }
  ]
}
```

サービスリンクロールの作成、編集、削除をIAM エンティティ (ユーザー、グループ、ロールなど) に許可する権限を構成する必要があります。詳細については、「IAM ユーザーガイド」の「サービスリンクロール権限」を参照してください。

Device Farm のサービスリンクロールの作成

モバイルテストプロジェクトの VPC 構成を提供する場合、サービスリンクロールを手動で作成する必要はありません。 AWS Management Console、、 AWS CLIまたは AWS API で最初の Device Farm リソースを作成すると、Device Farm によってサービスにリンクされたロールが作成されます。

このサービスリンクロールを削除した後で再度作成する必要が生じた場合は、同じ方法でアカウントにロールを再作成できます。Device Farm リソースを初めて作成すると、Device Farm がサービスリンクロールを再度作成します。

IAM コンソールを使用して、Device Farm ユースケースでサービスリンクロールを作成することもできます。 AWS CLI または AWS API で、サービス名を使用して`devicefarm.amazonaws.com`サービスにリンクされたロールを作成します。詳細については、「IAM ユーザーガイド」の「サービスにリンクされたロールの作成」を参照してください。このサービスリンクロールを削除しても、同じ方法でロールを再作成できます。

Device Farm のサービスリンクロールの編集

Device Farm では、 AWSServiceRoleForDeviceFarm サービスにリンクされたロールを編集することはできません。サービスリンクロールを作成した後は、多くのエンティティによってロールが参照される可能性があるため、ロール名を変更することはできません。ただし、IAM を使用したロールの説明の編集はできます。詳細については、「IAM ユーザーガイド」の「サービスリンクロールの編集」を参照してください。

Device Farm のサービスリンクロールの削除

サービスリンクロールが必要な機能またはサービスが不要になった場合には、そのロールを削除することをお勧めします。そうすることで、積極的にモニタリングまたは保守されていない未使用のエンティティを排除できます。ただし、手動で削除する前に、サービスリンクロールのリソースをクリーンアップする必要があります。

> **Note**
>
> リソースを削除する際に、Device Farm サービスがそのロールを使用している場合、削除が失敗することがあります。その場合は、数分待ってからオペレーションを再試行してください。

IAM を使用してサービスリンクロールを手動で削除するには

IAM コンソール、、または AWS API を使用して AWS CLI、 AWSServiceRoleForDeviceFarm サービスにリンクされた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「サービスリンクロールの削除」を参照してください。

Device Farm のサービスリンクロールがサポートされるリージョン

Device Farm は、サービスが利用可能なすべてのリージョンで、サービスリンクロールの使用をサポートします。詳細については、「AWS リージョンとエンドポイント」を参照してください。

Device Farm は、サービスを利用できるすべてのリージョンで、サービスリンクロールの使用をサポートしているわけではありません。 AWSServiceRoleForDeviceFarm ロールは、次のリージョンで使用できます。

リージョン名	リージョン識別子	Device Farm でのサポート
米国東部 (バージニア北部)	us-east-1	いいえ
米国東部 (オハイオ)	us-east-2	いいえ
米国西部 (北カリフォルニア)	us-west-1	いいえ
米国西部 (オレゴン)	us-west-2	はい
アジアパシフィック (ムンバイ)	ap-south-1	いいえ
アジアパシフィック (大阪)	ap-northeast-3	いいえ
アジアパシフィック (ソウル)	ap-northeast-2	いいえ
アジアパシフィック (シンガポール)	ap-southeast-1	いいえ

リージョン名	リージョン識別子	Device Farm でのサポート
アジアパシフィック (シドニー)	ap-southeast-2	いいえ
アジアパシフィック (東京)	ap-northeast-1	いいえ
カナダ (中部)	ca-central-1	いいえ
欧州 (フランクフルト)	eu-central-1	いいえ
欧州 (アイルランド)	eu-west-1	いいえ
欧州 (ロンドン)	eu-west-2	いいえ
欧州 (パリ)	eu-west-3	いいえ
南米 (サンパウロ)	sa-east-1	いいえ
AWS GovCloud (US)	us-gov-west-1	いいえ

前提条件

次のリストは、VPC-ENI 構成を作成する際に確認すべきいくつかの要件と提案を示しています。

- プライベートデバイスを AWS アカウントに割り当てる必要があります。
- サービスにリンクされたロールを作成するには、アクセス許可を持つ AWS アカウントユーザーまたはロールが必要です。Device Farm モバイルテスト機能で Amazon VPC エンドポイントを使用する場合、Device Farm は AWS Identity and Access Management (IAM) サービスにリンクされたロールを作成します。
- Device Farm は、`us-west-2` リージョン内の VPC にのみ接続できます。`us-west-2` リージョンに VPC がない場合は、作成する必要があります。次に、別のリージョンの VPC のリソースにアクセスするには、`us-west-2` リージョンの VPC と他のリージョンの VPC との間にピアリング接続を確立する必要があります。VPC のピアリングについては、「Amazon VPC ピアリングガイド」を参照してください。

接続を構成するときは、指定した VPC へのアクセス権を持つことを証明する必要があります。Device Farm では、特定の Amazon Elastic Compute Cloud (Amazon EC2) 権限を構成する必要があります。

- 使用する VPC では DNS 解決が必要です。
- VPC を作成したら、us-west-2 リージョン内の VPC に関する以下の情報が必要になります。
  - VPC ID
  - サブネット ID
  - セキュリティグループ ID
- Amazon VPC 接続は個々のプロジェクトベースで構成する必要があります。現時点では、1 つのプロジェクトで構成できる VPC 構成は 1 つだけです。VPC を構成すると、Amazon VPC は VPC 内にインターフェイスを作成し、指定されたサブネットとセキュリティグループに割り当てます。プロジェクトに関連するその後のセッションはすべて、構成された VPC 接続を使用します。
- VPC-ENI 構成をレガシー VPCE 機能と一緒に使用することはできません。
- VPC-ENI 構成を使用して既存のプロジェクトを更新しないことを強くお勧めします。既存のプロジェクトには、実行レベルで VPCE 設定が維持される可能性があるためです。代わりに、既存の VPCE 機能をすでに使用している場合は、すべての新しいプロジェクトに VPC-ENI を使用してください。

Amazon VPC への接続

Amazon VPC エンドポイントを使用するようにプロジェクトを構成および更新できます。VPC-ENI 構成は、個々のプロジェクトベースで構成されます。1 つのプロジェクトは常に 1 つの VPC-ENI エンドポイントしか持てません。プロジェクトの VPC アクセスを構成するには、次の情報を把握しておく必要があります:

- アプリケーションがそこでホストされている場合は us-west-2 の VPC ID、それ以外の場合は別のリージョンの他の us-west-2 VPC に接続する VPC ID。
- 接続に適用する適切なセキュリティグループ。
- 接続に関連付けられるサブネット。セッションが開始されると、使用可能な最大のサブネットが使用されます。VPC 接続の可用性を向上させるために、複数のサブネットを異なるアベイラビリティーゾーンに関連付けることをお勧めします。

VPC-ENI 構成を作成したら、次の手順に従ってコンソールまたは CLI を使用して詳細を更新できます。

Console

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. Device Farm ナビゲーションパネルで、[モバイルデバイスのテスト] を選択して、[プロジェクト] を選択します。
3. 「モバイルテストプロジェクト」で、リストからプロジェクトの名前を選択します。
4. [プロジェクト設定] を選択します。
5. 「仮想プライベートクラウド (VPC) 設定」セクションでは、VPC、Subnets、および Security Groups を変更できます。
6. [保存] を選択します。

CLI

以下の AWS CLI コマンドを使用して Amazon VPC を更新します。

```
$  aws devicefarm update-project \
--arn arn:aws:devicefarm:us-
west-2:111122223333:project:12345678-1111-2222-333-456789abcdef \
--vpc-config \
securityGroupIds=sg-02c1537701a7e3763,sg-005dadf9311efda25,\
subnetIds=subnet-09b1a45f9cac53717,subnet-09b1a45f9cac12345,\
vpcId=vpc-0238fb322af81a368
```

プロジェクトを作成するときに Amazon VPC を構成することもできます。

```
$  aws devicefarm create-project \
--name VPCDemo \
--vpc-config \
securityGroupIds=sg-02c1537701a7e3763,sg-005dadf9311efda25,\
subnetIds=subnet-09b1a45f9cac53717,subnet-09b1a45f9cac12345,\
vpcId=vpc-0238fb322af81a368
```

制限

VPC-ENI 機能には次の制限が適用されます。

- Device Farm プロジェクトの VPC 構成には、最大 5 つのセキュリティグループを指定できます。
- Device Farm プロジェクトの VPC 構成には、最大 8 つのサブネットを指定できます。
- Device Farm プロジェクトを VPC と連携するように構成する場合、提供できる最小のサブネットには、使用可能な IPv4 アドレスが少なくとも 5 つ必要です。
- パブリック IP アドレスは現時点ではサポートされていません。代わりに、Device Farm プロジェクトでプライベートサブネットを使用することをお勧めします。テスト中にパブリックインターネットアクセスが必要な場合は、ネットワークアドレス変換 (NAT) ゲートウェイを使用してください。パブリックサブネットで Device Farm プロジェクトを構成しても、テストにインターネットアクセスやパブリック IP アドレスは提供されません。
- サービス管理される ENI からの送信トラフィックのみがサポートされます。つまり、ENI は VPC からの未承諾のインバウンドリクエストを受信できません。

Device Farm での Amazon VPC エンドポイントサービスの使用 - レガシー (非推奨)

> **Warning**
>
> VPCE はレガシー機能と見なされるようになったため、プライベートエンドポイント接続には、このページで説明されている VPC-ENI 接続を使用することを強くお勧めします。VPC-ENI は、VPCE 接続方法と比較して、柔軟性と設定の簡素化、コスト効率の向上、メンテナンスオーバーヘッドの大幅な削減を実現します。

> **Note**
>
> Device Farm による Amazon VPC エンドポイントサービスの使用は構成済みプライベートデバイスをお使いのお客様に対してのみサポートされます。プライベートデバイスでこの機能を使用するために AWS アカウントを有効化するには、お問い合わせください。

Amazon Virtual Private Cloud (Amazon VPC) は、定義した仮想ネットワークで AWS リソースを起動するために使用できる AWS サービスです。VPC を使用すると、IP アドレス範囲、サブネット、ルートテーブル、ネットワークゲートウェイなどのネットワーク設定を制御できます。

Amazon VPC を使用して米国西部 (オレゴン) (us-west-2) AWS リージョンでプライベートアプリケーションをホストする場合、VPC と Device Farm の間にプライベート接続を確立できます。この接続により、Device Farm を使用し、パブリックインターネットを介して公開することなく、プライベートアプリケーションをテストすることができます。 AWS アカウントでこの機能をプライベートデバイスで使用できるようにするには、 にお問い合わせください。

お使いの VPC のリソースを Device Farm に接続するには、Amazon VPC コンソールを使用して、VPC エンドポイントサービスを作成します。このエンドポイントサービスでは、Device Farm VPC エンドポイントを介して VPC のリソースが Device Farm に提供されます。このエンドポイントサービスでは、インターネットゲートウェイ、ネットワークアドレス変換 (NAT) インスタンス、または VPN 接続を必要とせずに、信頼性の高いスケーラブルな接続性が Device Farm に提供されます。詳細については、「 AWS PrivateLink ガイド」の「VPC エンドポイントサービス (AWS PrivateLink) 」を参照してください。

> **Important**
>
> Device Farm VPC エンドポイント機能は、 AWS PrivateLink 接続を使用して VPC 内のプライベート内部サービスを Device Farm パブリック VPC に安全に接続するのに役立ちます。接続は安全でプライベートですが、そのセキュリティはお客様の AWS 認証情報の保護によって異なります。 AWS 認証情報が侵害された場合、攻撃者はサービスデータにアクセスして外部に公開する可能性があります。

Amazon VPC で VPC エンドポイントサービスを作成した後は、Device Farm コンソールを使用して、Device Farm で VPC エンドポイント構成を作成できます。このトピックでは、Device Farm で Amazon VPC 接続と VPC エンドポイント構成を作成する方法について説明します。

開始する前に

以下の情報は、サブネットを us-west-2a、us-west-2b、および us-west-2c といった各 Availability Zone で持つ、米国西部 (オレゴン) (us-west-2) リージョンの Amazon VPC ユーザー向けとなります。

Device Farm には、一緒に使用できる VPC エンドポイントサービスについての追加要件があります。Device Farm で作業するために VPC エンドポイントサービスを作成および構成する場合は、必ず次の要件を満たすオプションを選択してください。

- このサービスの Availability Zone は、us-west-2a、us-west-2b、および us-west-2c を含む必要があります。VPC エンドポイントサービスの Availability Zone は、エンドポイントサービスに関連付けられている Network Load Balancer によって決まります。VPC エンドポイントサービスにこれら 3 つのアベイラビリティーゾーンがすべて表示されない場合は、これら 3 つのゾーンを有効にするように Network Load Balancer を再作成してから、Network Load Balancer をエンドポイントサービスに再度関連付ける必要があります。
- エンドポイントサービス用に許可されたプリンシパルには、 Device Farm VPC エンドポイント (サービス ARN) の Amazon リソースネーム (ARN) が含まれる必要があります。エンドポイントサービスを作成した後、Device Farm VPC エンドポイントサービス ARN を許可リストに追加して、VPC エンドポイントサービスにアクセスするための Device Farm 権限を付与します。Device Farm VPC エンドポイントサービス ARN を取得するには、お問い合わせください。

また、VPC エンドポイントサービス作成時に [承認が必要] 設定が有効になっている場合は、Device Farm がエンドポイントサービスに送信する各接続リクエストを手動で承認する必要があります。既存のエンドポイントサービスに対して、この設定を変更するには、Amazon VPC コンソールでエンドポイントサービスを選択し、[アクション] を選択後、[エンドポイント承認設定を変更] を選択します。詳細については、「AWS PrivateLink ガイド」の「ロードバランサーと承認設定を変更する」を参照してください。

次のセクションでは、これらの要件を満たす Amazon VPC エンドポイントサービスを作成する方法について説明します。

ステップ 1: Network Load Balancer の作成

VPC と Device Farm の間でプライベート接続を確立する最初のステップは、Network Load Balancer を作成して、リクエストをターゲットグループにルーティングすることです。

New console

新しいコンソールを使用して Network Load Balancer を作成するには

1. https://console.aws.amazon.com/ec2/ で Amazon Elastic Compute Cloud (Amazon EC2) コンソールを開きます。
2. ナビゲーションペインの [ロードバランシング] で、[ロードバランサー] を選択します。

3. [ロードバランサーを作成] を選択します。
4. [Network Load Balancer] で、[作成] を選択します。
5. 「Network Load Balancer を作成する」ページの [基本構成] で、次の操作を行います。
   a. ロードバランサーの [名前] を入力します。
   b. [スキーム] では [内部] を選択します。
6. [ネットワークマッピング] で、次を行います:
   a. ターゲットグループの [VPC] を選択します。
   b. 次の [マッピング] を選択します:
      - `us-west-2a`
      - `us-west-2b`
      - `us-west-2c`
7. [リスナーとルーティング] で、[プロトコル] と [ポート] オプションを使用してターゲットグループを選択します。

> **Note**
> デフォルトでは、クロスアベイラビリティーゾーンロードバランシングは無効化されます。
> ロードバランサーは アベイラビリティーゾーン `us-west-2a`、`us-west-2b`、`us-west-2c` を使用するため、ターゲットをそれぞれのアベイラビリティゾーンに登録する必要があります。また、3 つのゾーンすべてにターゲットを登録しない場合は、クロスゾーンロードバランシングを有効にする必要があります。そうしないと、ロードバランサーが期待どおりに機能しない可能性があります。

8. [ロードバランサーを作成] を選択します。

Old console

古いコンソールを使用して Network Load Balancer を作成するには

1. https://console.aws.amazon.com/ec2/ で Amazon Elastic Compute Cloud (Amazon EC2) コンソールを開きます。
2. ナビゲーションペインの [ロードバランシング] で、[ロードバランサー] を選択します。

3. [ロードバランサーを作成] を選択します。
4. [Network Load Balancer] で、[作成] を選択します。
5. 「ロードバランサーを構成する」ページの [基本構成] で、次の操作を行います。

   a. ロードバランサーの [名前] を入力します。

   b. [スキーム] では [内部] を選択します。
6. [リスナー] で、ターゲットグループが使用している [プロトコル] と [ポート] を選択します。
7. [アベイラビリティゾーン] で次の操作を行います。

   a. ターゲットグループの [VPC] を選択します。

   b. 以下の [アベイラビリティゾーン] を選択します。

      - `us-west-2a`
      - `us-west-2b`
      - `us-west-2c`

   c. [次: セキュリティー設定を構成] を選択します。
8. (オプション) セキュリティを構成し、[次: ルーティングを構成] を選択します。
9. 「ルーティングを構成する」ページで、以下を実行します。

   a. [ターゲットグループ] で、[既存のターゲットグループ] を選択します。

   b. [名前] では、ターゲットグループを選択します。

   c. [次: ターゲットを登録] を選択します。
10. 「ターゲットを登録する」ページでターゲットを確認し、「次: 点検」を選択します。

    > **Note**
    >
    > デフォルトでは、クロスアベイラビリティーゾーンロードバランシングは無効化されます。
    > ロードバランサーは アベイラビリティーゾーン `us-west-2a`、`us-west-2b`、`us-west-2c` を使用するため、ターゲットをそれぞれのアベイラビリティゾーンに登録する必要があります。また、3 つのゾーンすべてにターゲットを登録しない場合は、クロスゾーンロードバランシングを有効にする必要があります。そうしないと、ロードバランサーが期待どおりに機能しない可能性があります。
11. ロードバランサー構成を点検し、[作成] を選択します。

ステップ 2: Amazon VPC エンドポイントサービスの作成

Network Load Balancer を作成したら、Amazon VPC コンソールを使用して、VPC にエンドポイントサービスを作成します。

1. Amazon VPC コンソール (https://console.aws.amazon.com/vpc/) を開きます。
2. [リージョン別リソース] で、[エンドポイントサービス] を選択します。
3. [エンドポイントサービスを作成] を選択します。
4. 次のいずれかを行います。

   - エンドポイントサービスで使用する Network Load Balancer がすでにある場合は、[利用可能なロードバランサー] でそれを選択して、ステップ 5 に進みます。
   - Network Load Balancer をまだ作成していない場合は、[新規ロードバランサーを作成] を選択します。Amazon EC2 コンソールが開きます。ステップ 3 から始まる「Network Load Balancer の作成」のステップに従い、Amazon VPC コンソールで次の手順に進みます。

5. [含まれるアベイラビリティーゾーン] では、`us-west-2a`、`us-west-2b`、および `us-west-2c` がリストに表示されていることを確認します。
6. エンドポイントサービスに送信される各接続リクエストを手動で承認または拒否しない場合は、[追加設定] で [承認が必要] チェックボックスのマークを外します。このチェックボックスのマークを外すと、エンドポイントサービスは受け取る各接続リクエストを自動的に承認します。
7. [作成] を選択します。
8. 新しいエンドポイントサービスを選択し、[プリンシパルを許可] の順に選択します。
9. エンドポイントサービスの許可リストに追加する Device Farm VPC エンドポイント (サービス ARN) の ARN を取得し、そのサービス ARN をサービスの許可リストに追加するには、私たちにお問い合わせください。
10. エンドポイントサービスの [詳細] タブで、サービスの名前 (サービス名) を書き留めます。この名前は、次のステップで VPC エンドポイント構成を作成する際に必要になります。

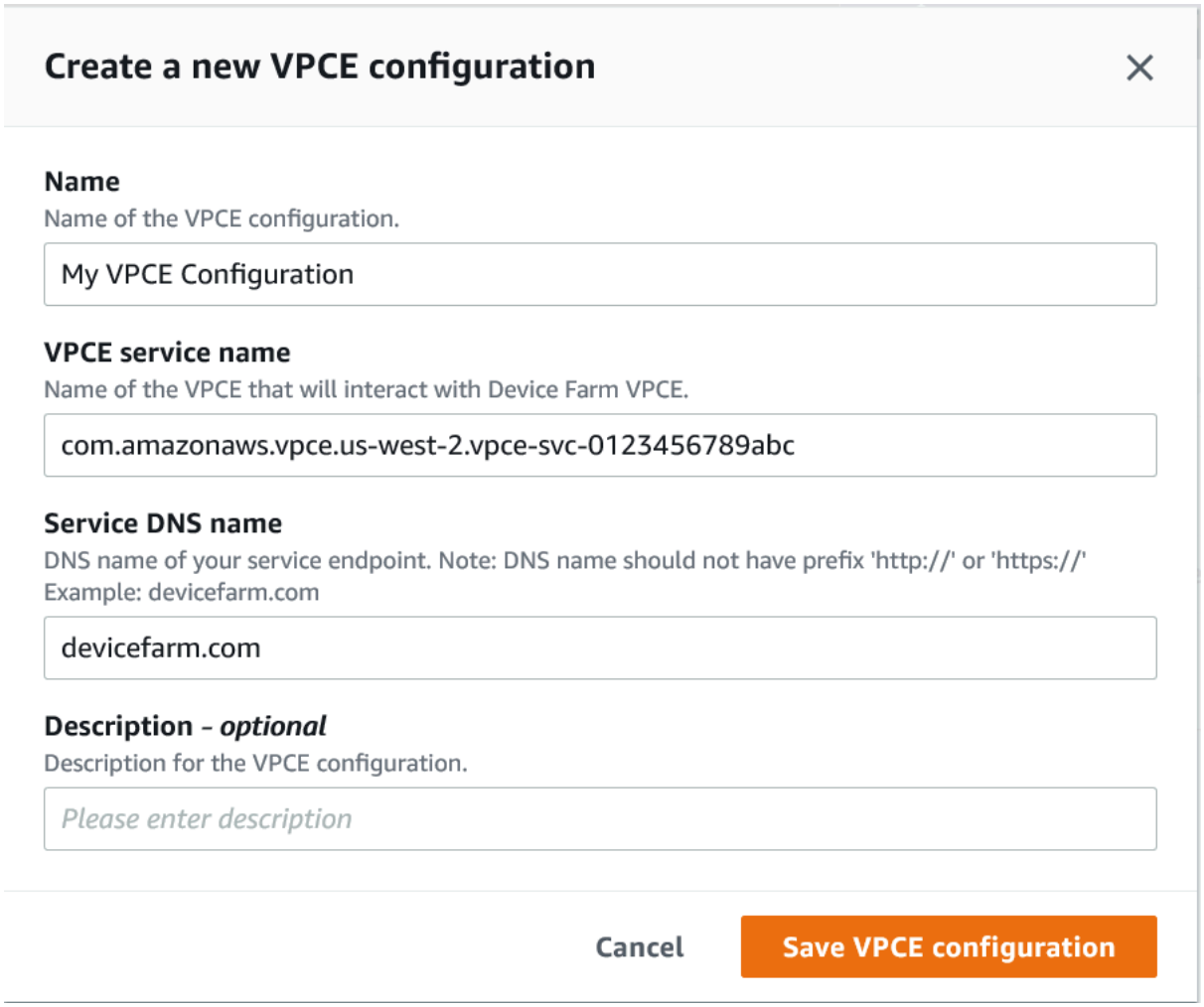
VPC エンドポイントサービスは現在、Device Farm により使用できます。

ステップ 3: Device Farm での VPC エンドポイント構成の作成

Amazon VPC でエンドポイントサービスを作成したら、Amazon VPC エンドポイント構成を Device Farm で作成できます。

1. https://console.aws.amazon.com/devicefarm で Device Farm コンソールにサインインします。
2. ナビゲーションペインで、[モバイルデバイスのテスト] を選択後、[プライベートデバイス] を選択します。
3. [VPCE 構成] を選択します。
4. [VPCE 構成を作成] を選択します。
5. [新規 VPCE 構成を作成] で、VPC エンドポイント構成の [名前] を入力します。
6. [VPCE サービス名] には、Amazon VPC エンドポイントサービスでメモした Amazon VPC エンドポイントサービスの名前 (サービス名) を入力します。名前は `com.amazonaws.vpce.us-west-2.vpce-svc-id` のようになります。
7. [Service DNS 名] には、テストするアプリケーションのサービス DNS 名 (例: `devicefarm.com`) を入力します。サービス DNS 名の前には `http` または `https` を指定しないでください。

   ドメイン名は、パブリックインターネットからはアクセスできません。また、VPC エンドポイントサービスにマッピングされるこの新規ドメイン名は、Amazon Route 53 によって生成され、Device Farm セッションでお客様専用として使用できます。
8. [保存] を選択します。



ステップ 4: テスト実行の作成

VPC エンドポイント構成を保存したら、その構成を使用してテスト実行またはリモートアクセスセッションを作成できます。詳細については、Device Farm でのテスト実行の作成またはセッションを作成するを参照してください。

AWS CloudTrail による AWS Device Farm API コールのロギング

AWS Device Farm は、ユーザー、ロール、または、 AWS Device Farm 内の AWS のサービスによる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。CloudTrail は、AWS Device Farm のすべての API コールをイベントとしてキャプチャします。キャプチャされた呼び出しには、AWS Device Farm コンソールからの呼び出しと、AWS Device Farm 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、AWS Device Farm のイベントなど、Amazon S3 バケットへの CloudTrail イベントの継続的な配信を有効にできます。証跡を構成しない場合でも、[イベント履歴] で CloudTrail コンソールの最新イベントを表示できます。CloudTrail で収集された情報を使用して、AWS Device Farm に対して行ったリクエスト、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。

CloudTrail の詳細については「AWS CloudTrail ユーザーガイド」を参照してください。

CloudTrail での AWS Device Farm 情報

AWS アカウントを作成すると、そのアカウントに対して CloudTrail が有効になります。AWS Device Farm でアクティビティが発生すると、そのアクティビティは [イベント履歴] で他の AWS サービスイベントとともに CloudTrail イベントに記録されます。AWS アカウントで最近のイベントを表示、検索、ダウンロードできます。詳細については、「CloudTrail イベント履歴でのイベントの表示」を参照してください。

AWS Device Farm のイベントなど、AWS アカウントのイベントを継続的に記録するには、証跡を作成します。証跡により、CloudTrail はログファイルを Amazon S3 バケットに配信できます。デフォルトでは、コンソールで作成した証跡がすべての AWS リージョンに適用されます。証跡は、AWS パーティションのすべてのリージョンからのイベントをログに記録し、指定した Amazon S3 バケットにログファイルを配信します。さらに、CloudTrail ログで収集したイベントデータをより詳細に分析し、それに基づいて対応するため、他の AWS サービスを構成できます。詳細については、次を参照してください:

- 証跡の作成のための概要
- CloudTrail がサポートするサービスと統合
- CloudTrail 用 Amazon SNS 通知の構成
- 「複数リージョンからの CloudTrail ログファイルの受信」および「複数アカウントからの CloudTrail ログファイルの受信」

AWS アカウントで CloudTrail のロギングを有効にすると、Device Farm アクションに行った API 呼び出しがログファイルに記録されます。Device Farm レコードは、他の AWS サービス記録と一緒にログファイルに記述されます。CloudTrail は、期間とファイルサイズに基づいて、新しいファイルをいつ作成して書き込むかを決定します。

Device Farm アクションはすべて、「AWS CLI リファレンス」および「Device Farm の自動化」に記録され、文書化されます。例えば、Device Farm で新しいプロジェクトまたは実行を作成する呼び出しを行うと、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、リクエストの生成者に関する情報が含まれます。アイデンティティ情報は、以下を判別するのに役立ちます:

- リクエストが、ルート認証情報と AWS Identity and Access Management (IAM) ユーザー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ユーザーまたはフェデレーションユーザーの一時的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 サービスによって行われたかどうか。

詳細については、「CloudTrail userIdentity エレメント」を参照してください。

AWS Device Farm ログファイルエントリの理解

「証跡」は構成として、指定した Amazon 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します。CloudTrail のログファイルは、単一か複数のログエントリを含みます。イベントは任意ソースからの単一リクエストを表し、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の情報を含みます。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、Device Farm `ListRuns` アクションを証明する CloudTrail ログエントリを表します:

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "Root",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:root",
        "accountId": "123456789012",
```

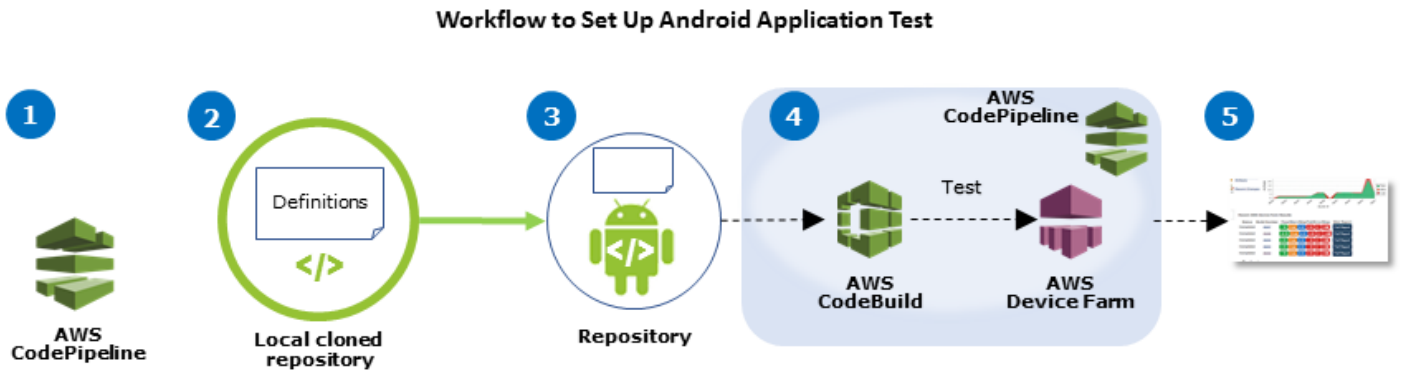
```
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
          "attributes": {
            "mfaAuthenticated": "false",
            "creationDate": "2015-07-08T21:13:35Z"
          }
        }
      },
      "eventTime":"2015-07-09T00:51:22Z",
      "eventSource": "devicefarm.amazonaws.com",
      "eventName":"ListRuns",
      "awsRegion":"us-west-2",
      "sourceIPAddress":"203.0.113.11",
      "userAgent":"example-user-agent-string",
      "requestParameters": {
        "arn":"arn:aws:devicefarm:us-west-2:123456789012:project:a9129b8c-
df6b-4cdd-8009-40a25EXAMPLE"},
        "responseElements": {
          "runs": [
            {
              "created": "Jul 8, 2015 11:26:12 PM",
              "name": "example.apk",
              "completedJobs": 2,
              "arn": "arn:aws:devicefarm:us-west-2:123456789012:run:a9129b8c-
df6b-4cdd-8009-40a256aEXAMPLE/1452d105-e354-4e53-99d8-6c993EXAMPLE",
              "counters": {
                "stopped": 0,
                "warned": 0,
                "failed": 0,
                "passed": 4,
                "skipped": 0,
                "total": 4,
                "errored": 0
              },
              "type": "BUILTIN_FUZZ",
              "status": "RUNNING",
              "totalJobs": 3,
              "platform": "ANDROID_APP",
              "result": "PENDING"
            },
            ... additional entries ...
          ]
        }
      }
```

```
    }
  ]
}
```

CodePipeline テストステージでの AWS Device Farm の使用

AWS CodePipeline を使用すると、Device Farm で構成したモバイルアプリケーションテストを AWS マネージド型の自動リリースパイプラインに組み込めます。オンデマンドで、スケジュールに従って、または継続的な統合フローの一部として、テスト実行のためのパイプラインを構成できます。

以下の図に示しているのは、プッシュがリポジトリにコミットされるたびに Android アプリケーションがビルドされてテストされる、継続的な統合フローです。このパイプライン構成を作成するには、「チュートリアル: GitHub にプッシュするときの Android アプリケーションのビルドとテスト」を参照してください。



1. 構成する	2. 定義を追加する	3. プッシュする	4. ビルドとテストを行う	5. レポートを行う
パイプラインリソースを構成する	パッケージにビルドとテストの定義を追加する	リポジトリにパッケージをプッシュする	アプリケーションをビルドし、自動的に開始されるビルド出力アーティファクトをテストする	テスト結果を表示する

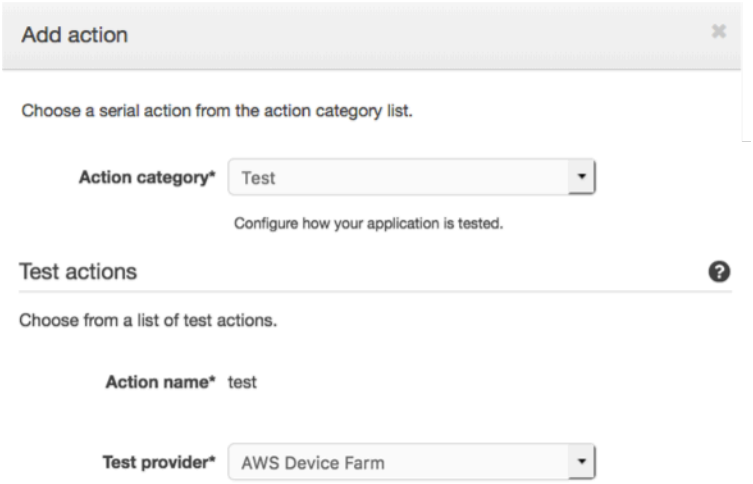
コンパイルされたアプリケーション (iOS の `.ipa` ファイルや Android の `.apk` ファイルなど) をソースとして継続的にテストするパイプラインを構成する方法については、「チュートリアル: .ipa ファイルを Amazon S3 バケットにアップロードするたびに iOS アプリケーションをテストする」を参照してください。

Device Farm テストを使用するために CodePipeline を構成する

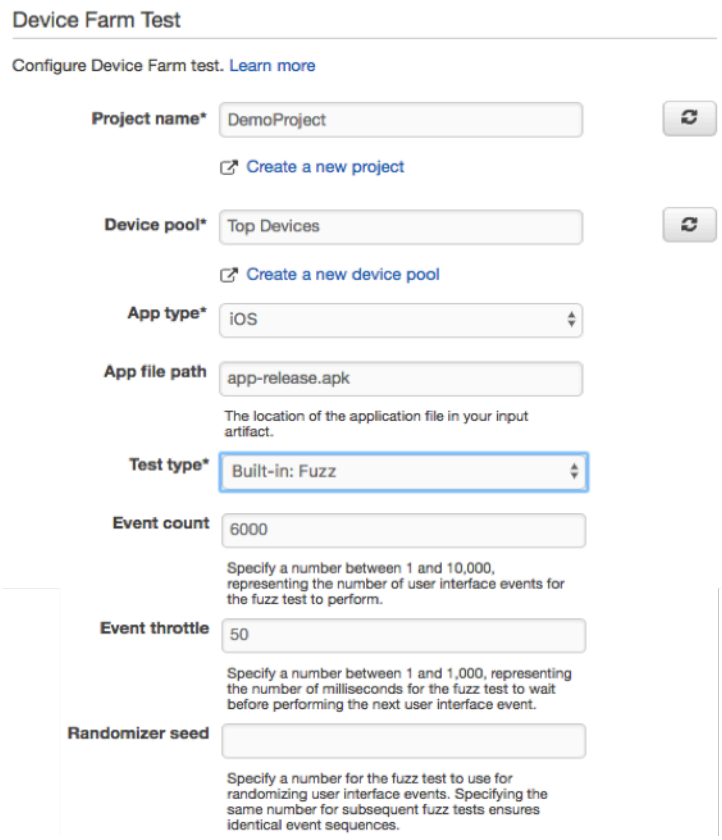
以下のステップは、Device Farm プロジェクトの構成とパイプラインの作成を完了していることが前提となります。テストステージで、テスト定義とコンパイルされたアプリケーションパッケージファイルを含む入力アーティファクトを受け取るように、パイプラインを構成する必要があります。テストステージの入力アーティファクトとしては、パイプラインで構成したソースステージまたはビルドステージの出力アーティファクトを使用できます。

Device Farm テスト実行を CodePipeline テストアクションとして構成するには

1. AWS Management Console にサインインして、https://console.aws.amazon.com/codepipeline/ で CodePipeline コンソールを開きます。
2. アプリケーションリリースのパイプラインを選択します。
3. テストステージパネルで、鉛筆アイコンを選択してから、[アクション] を選択します。
4. [アクションを追加] パネルの [アクションカテゴリー] で、[テスト] を選択します。
5. [アクション名] に名前を入力します。
6. [テストプロバイダー] で、[AWS Device Farm] を選択します。

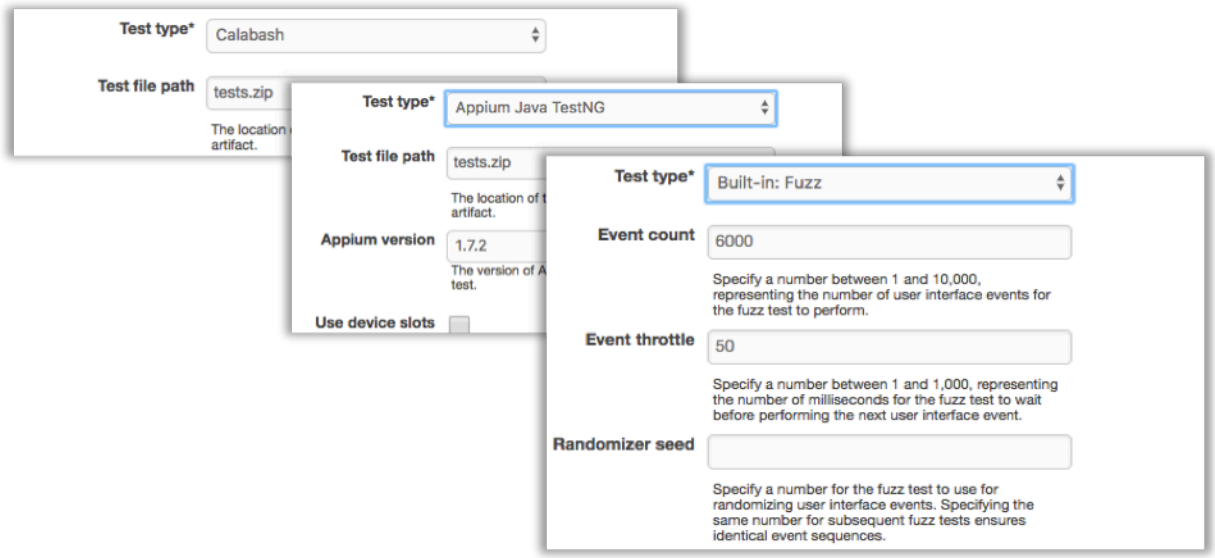


7. [プロジェクト名] で、既存の Device Farm プロジェクトを選択するか、[新規プロジェクトを作成] を選択します。
8. [デバイスプール] で、既存のデバイスプールを選択するか、[新規デバイスプールを作成] を選択します。デバイスプールを作成する場合は、一連のテストデバイスを選択する必要があります。
9. [アプリケーションタイプ] で、アプリケーションのプラットフォームを選択します。

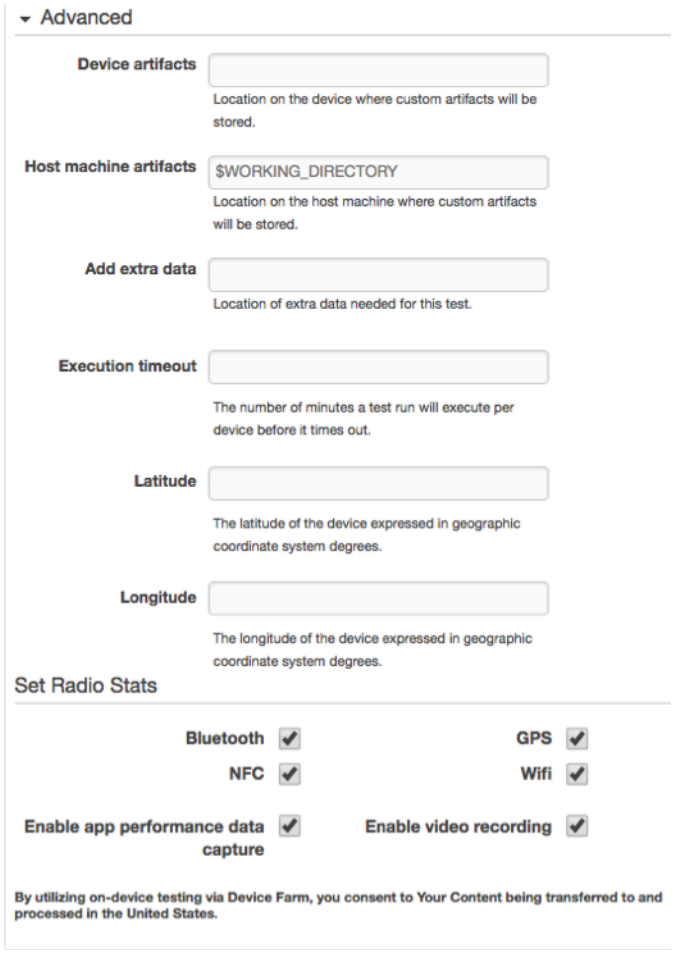


10. [アプリケーションファイルパス] にコンパイルされたアプリケーションパッケージのパスを入力します。パスは、テストの入力アーティファクトのルートに関連します。
11. [テストタイプ] で以下のいずれかを実行します:

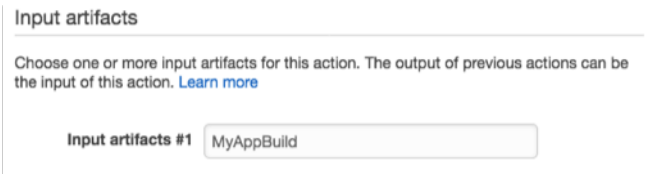
    - ビルトイン Device Farm テストのいずれかを使用している場合は、Device Farm プロジェクトで構成されているテストのタイプを選択します。
    - Device Farm のビルトインテストのいずれも使用していない場合は、[テストファイルパス] にテスト定義ファイルのパスを入力します。パスは、テストの入力アーティファクトのルートに関連します。



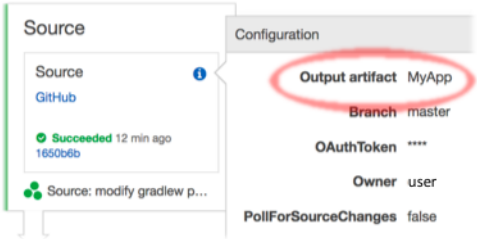
12. 残りのフィールドにはテストおよびアプリケーションタイプに適した構成を入力します。
13. (オプション) [詳細] で、テスト実行の詳細な構成を伝えます。



14. [入力アーティファクト] で、パイプラインのテストステージの前にあるステージの出力アーティファクトに一致する入力アーティファクトを選択します。



CodePipeline コンソールでは、パイプライン図の情報アイコンの上にカーソルを置くことで、各ステージの出力アーティファクトの名前を見つけられます。パイプラインでアプリケーションを [ソース] ステージから直接テストする場合は、[MyApp] を選択します。パイプラインが ビルドステージを含む場合は、[MyAppBuild] を選択します。



15. パネルの下部で、[アクションを追加] を選択します。
16. CodePipeline ペインで、[パイプラインの変更を保存]、[変更を保存] の順に選択します。
17. 変更を送信してパイプラインのビルドを開始するには、[変更をリリース]、[リリース] の順に選択します。

AWS Device Farm 用 AWS CLI リファレンス

Device Farm コマンドを実行する AWS Command Line Interface (AWS CLI) を使用するには、「AWS CLI AWS Device Farm 用リファレンス」を参照してください。

AWS CLI に関する一般情報については、「AWS Command Line Interface ユーザーガイド」と「AWS CLI コマンドリファレンス」を参照してください。

AWS Device Farm 用 Windows PowerShell リファレンス

Windows PowerShell を使用して Device Farm コマンドを実行するには、「AWS Tools for Windows PowerShellコマンドレットリファレンス」内の「Device Farm コマンドレットリファレンス」を参照してください。詳細については、「AWS Tools for Windows PowerShell ユーザーガイド」内の「AWS Tools for Windows PowerShell のセットアップ」を参照してください。

AWS Device Farm の自動化

プログラムによる Device Farm へのアクセスは、実行のスケジュールや実行、スイート、またはテスト用のアーティファクトのダウンロードなど、実行する必要がある一般的なタスクを自動化するための強力な方法です。AWS SDK と AWS CLI は、そのための手段を提供します。

AWS SDK は、Device Farm、Amazon S3 などを含むすべての AWS のサービスへのアクセスを提供します。詳細については、次を参照する

- AWS ツールと SDK
- AWS Device Farm API リファレンス

例: AWS SDK を使用した Device Farm 実行の開始とアーティファクトの収集

次の例では、AWS SDK を使用して Device Farm での作業を可能にする方法についてのデモを最初から最後まで提供します。この例では、次のような処理を実行します。

- テストパッケージとアプリケーションパッケージを Device Farm にアップロードする
- テスト実行を開始し、その完了 (または失敗) を待つ
- テストスイートによって生成されたすべてのアーティファクトをダウンロードする

この例は、HTTP と対話するサードパーティーの `requests` パッケージに依存しています。

```
import boto3
import os
import requests
import string
import random
import time
import datetime
import time
import json

# The following script runs a test through Device Farm
#
# Things you have to change:
config = {
```

```
    # This is our app under test.
    "appFilePath":"app-debug.apk",
    "projectArn": "arn:aws:devicefarm:us-
west-2:111122223333:project:1b99bcff-1111-2222-ab2f-8c3c733c55ed",
    # Since we care about the most popular devices, we'll use a curated pool.
    "testSpecArn":"arn:aws:devicefarm:us-west-2::upload:101e31e8-12ac-11e9-ab14-
d663bd873e83",
    "poolArn":"arn:aws:devicefarm:us-west-2::devicepool:082d10e5-d7d7-48a5-ba5c-
b33d66efa1f5",
    "namePrefix":"MyAppTest",
    # This is our test package. This tutorial won't go into how to make these.
    "testPackage":"tests.zip"
}

client = boto3.client('devicefarm')

unique =
 config['namePrefix']+"-"+(datetime.date.today().isoformat())+(''.join(random.sample(string.asc

print(f"The unique identifier for this run is going to be {unique} -- all uploads will
 be prefixed with this.")

def upload_df_file(filename, type_, mime='application/octet-stream'):
    response = client.create_upload(projectArn=config['projectArn'],
        name = (unique)+"_"+os.path.basename(filename),
        type=type_,
        contentType=mime
        )
    # Get the upload ARN, which we'll return later.
    upload_arn = response['upload']['arn']
    # We're going to extract the URL of the upload and use Requests to upload it
    upload_url = response['upload']['url']
    with open(filename, 'rb') as file_stream:
        print(f"Uploading {filename} to Device Farm as {response['upload']['name']}...
 ",end='')
        put_req = requests.put(upload_url, data=file_stream, headers={"content-
type":mime})
        print(' done')
        if not put_req.ok:
            raise Exception("Couldn't upload, requests said we're not ok. Requests
 says: "+put_req.reason)
    started = datetime.datetime.now()
    while True:
```

```
        print(f"Upload of {filename} in state {response['upload']['status']} after
 "+str(datetime.datetime.now() - started))
        if response['upload']['status'] == 'FAILED':
            raise Exception("The upload failed processing. DeviceFarm says reason
 is: \n"+(response['upload']['message'] if 'message' in response['upload'] else
 response['upload']['metadata']))
        if response['upload']['status'] == 'SUCCEEDED':
            break
        time.sleep(5)
        response = client.get_upload(arn=upload_arn)
    print("")
    return upload_arn

our_upload_arn = upload_df_file(config['appFilePath'], "ANDROID_APP")
our_test_package_arn = upload_df_file(config['testPackage'],
 'APPIUM_PYTHON_TEST_PACKAGE')
print(our_upload_arn, our_test_package_arn)
# Now that we have those out of the way, we can start the test run...
response = client.schedule_run(
    projectArn = config["projectArn"],
    appArn = our_upload_arn,
    devicePoolArn = config["poolArn"],
    name=unique,
    test = {
        "type":"APPIUM_PYTHON",
        "testSpecArn": config["testSpecArn"],
        "testPackageArn": our_test_package_arn
        }
    )
run_arn = response['run']['arn']
start_time = datetime.datetime.now()
print(f"Run {unique} is scheduled as arn {run_arn} ")

try:

    while True:
        response = client.get_run(arn=run_arn)
        state = response['run']['status']
        if state == 'COMPLETED' or state == 'ERRORED':
            break
        else:
            print(f" Run {unique} in state {state}, total time
 "+str(datetime.datetime.now()-start_time))
            time.sleep(10)
```

```
except:
    # If something goes wrong in this process, we stop the run and exit.

    client.stop_run(arn=run_arn)
    exit(1)
print(f"Tests finished in state {state} after "+str(datetime.datetime.now() -
 start_time))
# now, we pull all the logs.
jobs_response = client.list_jobs(arn=run_arn)
# Save the output somewhere. We're using the unique value, but you could use something
 else
save_path = os.path.join(os.getcwd(), unique)
os.mkdir(save_path)
# Save the last run information
for job in jobs_response['jobs'] :
    # Make a directory for our information
    job_name = job['name']
    os.makedirs(os.path.join(save_path, job_name), exist_ok=True)
    # Get each suite within the job
    suites = client.list_suites(arn=job['arn'])['suites']
    for suite in suites:
        for test in client.list_tests(arn=suite['arn'])['tests']:
            # Get the artifacts
            for artifact_type in ['FILE','SCREENSHOT','LOG']:
                artifacts = client.list_artifacts(
                    type=artifact_type,
                    arn = test['arn']
                )['artifacts']
                for artifact in artifacts:
                    # We replace : because it has a special meaning in Windows & macos
                    path_to = os.path.join(save_path, job_name, suite['name'],
 test['name'].replace(':','_') )
                    os.makedirs(path_to, exist_ok=True)
                    filename =
 artifact['type']+"_"+artifact['name']+"."+artifact['extension']
                    artifact_save_path = os.path.join(path_to, filename)
                    print("Downloading "+artifact_save_path)
                    with open(artifact_save_path, 'wb') as fn,
 requests.get(artifact['url'],allow_redirects=True) as request:
                        fn.write(request.content)
                    #/for artifact in artifacts
                #/for artifact type in []
            #/ for test in ()[]
        #/ for suite in suites
```

```
    #/ for job in _[]
# done
print("Finished")
```

Device Farm エラーのトラブルシューティング

このセクションでは、Device Farm に関する一般的な問題を修正するのに役立つエラーメッセージと手順を示します。

トピック

AWS Device Farm の Android アプリケーションテストのトラブルシューティング

次のトピックでは、Android アプリケーションテストのアップロード中に発生するエラーメッセージを挙げ、各エラーを解決するための推奨回避策を伝えます。

> **Note**
>
> 以下の手順は Linux x86_64 および Mac を対象にしています。

ANDROID_APP_UNZIP_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> アプリケーションを開けませんでした。ファイルが有効であることを確認して、もう一度お試しください。

エラーなしでアプリケーションパッケージを解凍できることを確かめてください。次の例では、パッケージ名は app-debug.apk です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip app-debug.apk
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   有効な Android アプリケーションパッケージでは、次のような出力が生成されます:

   ```
   .
   |-- AndroidManifest.xml
   |-- classes.dex
   |-- resources.arsc
   |-- assets (directory)
   |-- res (directory)
   `-- META-INF (directory)
   ```

   詳細については、「AWS Device Farm での Android テストによる作業」を参照してください。

ANDROID_APP_AAPT_DEBUG_BADGING_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> アプリケーションに関する情報を抽出できませんでした。`aapt debug badging <path to your test package>` コマンドを実行してアプリケーションが有効であることを確認し、コマンドがエラーを出力しなくなってからもう一度試してください。

アップロード検証プロセス中に、AWS Device Farm は `aapt debug badging <path to your package>` コマンドの出力から情報を解析します。

Android アプリケーションでこのコマンドを正常に実行できることを確かめてください。次の例では、パッケージ名は app-debug.apk です。

- アプリケーションパッケージを作業ディレクトリにコピーし、次にコマンドを実行します:

  ```
  $ aapt debug badging app-debug.apk
  ```

  有効な Android アプリケーションパッケージでは、次のような出力が生成されます:

  ```
  package: name='com.amazon.aws.adf.android.referenceapp' versionCode='1'
   versionName='1.0' platformBuildVersionName='5.1.1-1819727'
  sdkVersion:'9'
  application-label:'ReferenceApp'
  application: label='ReferenceApp' icon='res/mipmap-mdpi-v4/ic_launcher.png'
  application-debuggable
  launchable-activity:
   name='com.amazon.aws.adf.android.referenceapp.Activities.MainActivity'
   label='ReferenceApp' icon=''
  uses-feature: name='android.hardware.bluetooth'
  uses-implied-feature: name='android.hardware.bluetooth' reason='requested
   android.permission.BLUETOOTH permission, and targetSdkVersion > 4'
  main
  supports-screens: 'small' 'normal' 'large' 'xlarge'
  supports-any-density: 'true'
  locales: '--_--'
  densities: '160' '213' '240' '320' '480' '640'
  ```

  詳細については、「AWS Device Farm での Android テストによる作業」を参照してください。

ANDROID_APP_PACKAGE_NAME_VALUE_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> アプリケーション内にパッケージ名の値が見つかりませんでした。`aapt debug badging <path to your test package>` コマンドを実行してアプリケーションが有効であることを確認し、キーワード "package: name" の後にパッケージ名の値を見つけてからもう一度試して下さい。

アップロード検証プロセス中に、AWS Device Farm は `aapt debug badging <path to your package>` コマンドの出力からパッケージ名の値を解析します。

Android アプリケーションでこのコマンドを実行でき、パッケージ名の値を正常に見つけられることを確かめてください。次の例では、パッケージ名は app-debug.apk です。

- アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

  ```
  $ aapt debug badging app-debug.apk | grep "package: name="
  ```

  有効な Android アプリケーションパッケージでは、次のような出力が生成されます:

  ```
  package: name='com.amazon.aws.adf.android.referenceapp' versionCode='1'
   versionName='1.0' platformBuildVersionName='5.1.1-1819727'
  ```

  詳細については、「AWS Device Farm での Android テストによる作業」を参照してください。

ANDROID_APP_SDK_VERSION_VALUE_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> アプリケーション内に SDK バージョンの値が見つかりませんでした。`aapt debug badging <path to your test package>` コマンドを実行してアプリケーションが有効

であることを確認し、キーワード sdkVersion の後ろに SDK バージョンの値を見つけた後にもう一度試して下さい。

アップロード検証プロセス中に、AWS Device Farm は `aapt debug badging <path to your package>` コマンドの出力から SDK バージョンの値を解析します。

Android アプリケーションでこのコマンドを実行でき、パッケージ名の値を正常に見つけられることを確かめてください。次の例では、パッケージ名は app-debug.apk です。

- アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

  ```
  $ aapt debug badging app-debug.apk | grep "sdkVersion"
  ```

  有効な Android アプリケーションパッケージでは、次のような出力が生成されます:

  ```
  sdkVersion:'9'
  ```

  詳細については、「AWS Device Farm での Android テストによる作業」を参照してください。

ANDROID_APP_AAPT_DUMP_XMLTREE_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> 有効な AndroidManifest.xml がアプリケーション内に見つかりませんでした。コマンド `aapt dump xmltree <path to your test package> AndroidManifest.xml` を実行してテストパッケージが有効であることを確認し、コマンドがエラーを出力しなくなってからもう一度試してください。

アップロード検証プロセス中に、AWS Device Farm はコマンド `aapt dump xmltree <path to your package> AndroidManifest.xml` を使用してパッケージに含まれる XML ファイルの XML 解析ツリーから情報を解析します。

Android アプリケーションでこのコマンドを正常に実行できることを確認してください。次の例では、パッケージ名は app-debug.apk です。

- アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

  ```
  $ aapt dump xmltree app-debug.apk. AndroidManifest.xml
  ```

  有効な Android アプリケーションパッケージでは、次のような出力が生成されます:

  ```
  N: android=http://schemas.android.com/apk/res/android
    E: manifest (line=2)
      A: android:versionCode(0x0101021b)=(type 0x10)0x1
      A: android:versionName(0x0101021c)="1.0" (Raw: "1.0")
      A: package="com.amazon.aws.adf.android.referenceapp" (Raw:
   "com.amazon.aws.adf.android.referenceapp")
      A: platformBuildVersionCode=(type 0x10)0x16 (Raw: "22")
      A: platformBuildVersionName="5.1.1-1819727" (Raw: "5.1.1-1819727")
      E: uses-sdk (line=7)
        A: android:minSdkVersion(0x0101020c)=(type 0x10)0x9
        A: android:targetSdkVersion(0x01010270)=(type 0x10)0x16
      E: uses-permission (line=11)
        A: android:name(0x01010003)="android.permission.INTERNET" (Raw:
   "android.permission.INTERNET")
      E: uses-permission (line=12)
        A: android:name(0x01010003)="android.permission.CAMERA" (Raw:
   "android.permission.CAMERA")
  ```

  詳細については、「AWS Device Farm での Android テストによる作業」を参照してください。

ANDROID_APP_DEVICE_ADMIN_PERMISSIONS

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> アプリケーションにデバイス管理者権限が必要なことが見つかりました。aapt dump xmltree *<path to your test package>* AndroidManifest.xml コマンドを実行して権限が必要ないことを確認して、キーワード

android.permission.BIND_DEVICE_ADMIN が出力に含まれていないことを確かめてからもう一度試してください。

アップロード検証プロセス中に、AWS Device Farm はコマンド `aapt dump xmltree <path to your package> AndroidManifest.xml` を使用してパッケージに含まれる XML ファイルの XML 解析ツリーから権限情報を解析します。

アプリケーションにデバイス管理者権限が必要ないことを確認してください。次の例では、パッケージ名は app-debug.apk です。

- アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

```
$ aapt dump xmltree app-debug.apk AndroidManifest.xml
```

次のような出力が生まれます:

```
N: android=http://schemas.android.com/apk/res/android
  E: manifest (line=2)
    A: android:versionCode(0x0101021b)=(type 0x10)0x1
    A: android:versionName(0x0101021c)="1.0" (Raw: "1.0")
    A: package="com.amazonaws.devicefarm.android.referenceapp" (Raw:
 "com.amazonaws.devicefarm.android.referenceapp")
    A: platformBuildVersionCode=(type 0x10)0x16 (Raw: "22")
    A: platformBuildVersionName="5.1.1-1819727" (Raw: "5.1.1-1819727")
    E: uses-sdk (line=7)
      A: android:minSdkVersion(0x0101020c)=(type 0x10)0xa
      A: android:targetSdkVersion(0x01010270)=(type 0x10)0x16
    E: uses-permission (line=11)
      A: android:name(0x01010003)="android.permission.INTERNET" (Raw:
 "android.permission.INTERNET")
    E: uses-permission (line=12)
      A: android:name(0x01010003)="android.permission.CAMERA" (Raw:
 "android.permission.CAMERA")
        ……
```

Android アプリケーションが有効な場合、出力に以下は含まれません: `A: android:name(0x01010003)="android.permission.BIND_DEVICE_ADMIN" (Raw: "android.permission.BIND_DEVICE_ADMIN")`。

詳細については、「AWS Device Farm での Android テストによる作業」を参照してください。

Android アプリケーションの特定ウィンドウに真っ白または真っ黒の画面が表示される

Android アプリケーションをテストしていて、テストにおける Device Farm のビデオ録画でアプリケーション内の特定ウィンドウが真っ黒の画面となる場合は、アプリケーションが Android の FLAG_SECURE 機能を使用している可能性があります。このフラグ (Android の公式ドキュメンテーションに記載) は、画面記録ツールがアプリケーションの特定ウィンドウに記録を行わないようにするため使用されます そのため、Device Farm の画面記録機能 (自動化テストとリモートアクセステストの両方) により、このフラグを使用するアプリケーションウィンドウで真っ黒の画面が表示されることがあります。

このフラグは、開発者がログインページなどの機密情報を含むアプリケーション内のページによく使用します。ログインページなどの特定ページでアプリケーションの画面が真っ黒になる場合は、開発者と協力して、テストにこのフラグを使用しないアプリケーションのビルドを入手してください。

また、Device Farm は、このフラグが設定されているアプリケーションウィンドウに引き続き双方向対応できることに注意してください。そのため、アプリケーションのログインページが真っ黒の画面になっても、認証情報を入力してアプリケーションにログインできます (これにより、FLAG_SECURE フラグでブロックされていないページを表示できます)。

AWS Device Farm での Appium Java JUnit テストのトラブルシューティング

以下のトピックでは、Appium Java JUnit テストのアップロード中に発生するエラーメッセージを挙げ、各エラーを解決するための推奨回避策を伝えます。

> **Note**
>
> 以下の手順は Linux x86_64 および Mac を対象にしています。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_PACKAGE_UNZIP_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テスト ZIP ファイルを開けませんでした。ファイルが有効であることを確認してから、もう一度お試しください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージ名は zip-with-dependencies.zip です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍すると、次のコマンドを実行すれば作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   有効な Appium Java JUnit パッケージでは、次のような出力が生成されます:

   ```
   .
   |— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |— zip-with-dependencies.zip (this .zip file contains all of the items)
   `— dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
         |— com.some-dependency.bar-4.1.jar
         |— com.another-dependency.thing-1.0.jar
         |— joda-time-2.7.jar
         `— log4j-1.2.14.jar
   ```

   詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_DEPENDENCY_DIR_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> dependency-jars のディレクトリがテストパッケージ内に見つかりませんでした。テストパッケージを解凍し、dependency-jars ディレクトリがパッケージ内にあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip zip-with-dependencies.zip
```

2. 正常にパッケージを解凍すると、次のコマンドを実行すれば作業ディレクトリのツリー構造を見つけることができます:

```
$ tree .
```

Appium Java JUnit パッケージが有効な場合は、作業ディレクトリ内に *dependency-jars* ディレクトリがあります:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_JAR_MISSING_IN_DEPENDENCY_DI

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> dependency-jars ディレクトリツリー内に JAR ファイルが見つかりませんでした。テストパッケージを解凍してから dependency-jars ディレクトリを開き、少なくとも 1 つの JAR ファイルがディレクトリにあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip zip-with-dependencies.zip
```

2. 正常にパッケージを解凍すると、次のコマンドを実行すれば作業ディレクトリのツリー構造を見つけることができます:

```
$ tree .
```

Appium Java JUnit パッケージが有効な場合は、*dependency-jars* ディレクトリ内に少なくとも 1 つの *jar* ファイルがあります:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_TESTS_JAR_FILE_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テストパッケージ内に *-tests.jar ファイルが見つかりませんでした。テストパッケージを解凍し、パッケージ内に少なくとも 1 つの *-tests.jar ファイルがあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍すると、次のコマンドを実行すれば作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Java JUnit パッケージが有効な場合は、この例の *acme-android-appium-1.0-SNAPSHOT-tests.jar* のような *jar* ファイルが少なくとも 1 つあります。ファイルの名前は異なる場合がありますが、*-tests.jar* で終わります。

   ```
   .
   |– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |– zip-with-dependencies.zip (this .zip file contains all of the items)
   `– dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
         |– com.some-dependency.bar-4.1.jar
         |– com.another-dependency.thing-1.0.jar
         |– joda-time-2.7.jar
         `– log4j-1.2.14.jar
   ```

   詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TESTS_JA

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テスト JAR ファイル内にクラスファイルが見つかりませんでした。テストパッケージを解凍してから、テスト JAR ファイルをアンジャーし、JAR ファイル内に少なくとも 1 つのクラスファイルがあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍すると、次のコマンドを実行すれば作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   この例の *acme-android-appium-1.0-SNAPSHOT-tests.jar* のような jar ファイルが少なくとも 1 つあります。ファイルの名前は異なる場合がありますが、*-tests.jar* で終わります。

   ```
   .
   |— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |— zip-with-dependencies.zip (this .zip file contains all of the items)
   `— dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
         |— com.some-dependency.bar-4.1.jar
         |— com.another-dependency.thing-1.0.jar
         |— joda-time-2.7.jar
         `— log4j-1.2.14.jar
   ```

3. ファイルを正常に抽出したら、次のコマンドを実行すれば作業ディレクトリツリーに少なくとも 1 つのクラスがあるはずです:

```
$ tree .
```

次のような出力が表示されます:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
everything built from the ./src/test directory)
|- one-class-file.class
|- folder
|    `-another-class-file.class
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_JUNIT_VERSION_VALUE_UNKNOWN

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> JUnit のバージョン値が見つかりませんでした。テストパッケージを解凍し、dependency-jars ディレクトリを開き、JUnit JAR ファイルがディレクトリ内にあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip zip-with-dependencies.zip
```

2. 正常にパッケージを解凍すると、次のコマンドを実行すれば、作業ディレクトリのツリー構造を見つけることができます:

```
tree .
```

出力は次のようになります:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
    |— junit-4.10.jar
    |— com.some-dependency.bar-4.1.jar
    |— com.another-dependency.thing-1.0.jar
    |— joda-time-2.7.jar
    `— log4j-1.2.14.jar
```

Appium Java JUnit パッケージが有効な場合は、この例の jar ファイル *junit-4.10.jar* のような JUnit 依存関係ファイルがあります。名前はキーワード「*junit*」とバージョン番号からなります。この例では 4.10 です。

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_INVALID_JUNIT_VERSION

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> JUnit のバージョンが、サポート対象の最小バージョン 4.10 よりも低いことがわかりました。JUnit のバージョンを変更して、もう一度試してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip zip-with-dependencies.zip
```

2. 正常にパッケージを解凍すると、次のコマンドを実行すれば作業ディレクトリのツリー構造を見つけることができます:

```
$ tree .
```

この例の *junit-4.10.jar* のような JUnit 依存関係ファイルとそのバージョン番号 (この例では 4.10) が見つかります:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— junit-4.10.jar
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

> **Note**
>
> テストパッケージで指定された JUnit のバージョンが、サポート対象の最小バージョン 4.10 よりも低い場合、テストが正しく実行されないことがあります。

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

AWS Device Farm での Appium Java JUnit ウェブアプリケーションテストのトラブルシューティング

次のトピックでは、Appium Java JUnit ウェブアプリケーションテストのアップロード中に発生するエラーメッセージを挙げ、各エラーを解決するための推奨回避策を伝えます。Device Farm と Appium の使用の詳細については、「the section called “Appium”」を参照してください。

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_UNZIP_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テスト ZIP ファイルを開けませんでした。ファイルが有効であることを確認してから、もう一度お試しください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   有効な Appium Java JUnit パッケージでは、次のような出力が生成されます:

   ```
   .
   |— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |— zip-with-dependencies.zip (this .zip file contains all of the items)
   `— dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
   ```

```
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_DEPENDENCY_DIR_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> dependency-jars のディレクトリがテストパッケージ内に見つかりませんでした。テストパッケージを解凍し、dependency-jars ディレクトリがパッケージ内にあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Java JUnit パッケージが有効な場合は、作業ディレクトリ内に *dependency-jars* ディレクトリがあります:

   ```
   .
   |– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |– zip-with-dependencies.zip (this .zip file contains all of the items)
   `– dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
        |– com.some-dependency.bar-4.1.jar
   ```

```
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_JAR_MISSING_IN_DEPENDEN

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> dependency-jars ディレクトリツリー内に JAR ファイルが見つかりませんでした。テストパッケージを解凍してから dependency-jars ディレクトリを開き、少なくとも 1 つの JAR ファイルがディレクトリにあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Java JUnit パッケージが有効な場合は、*dependency-jars* ディレクトリ内に少なくとも 1 つの *jar* ファイルがあります:

   ```
   .
   |– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |– zip-with-dependencies.zip (this .zip file contains all of the items)
   `– dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
         |– com.some-dependency.bar-4.1.jar
         |– com.another-dependency.thing-1.0.jar
   ```

```
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_TESTS_JAR_FILE_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テストパッケージ内に *-tests.jar ファイルが見つかりませんでした。テストパッケージを解凍し、パッケージ内に少なくとも 1 つの *-tests.jar ファイルがあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Java JUnit パッケージが有効な場合は、この例の *acme-android-appium-1.0-SNAPSHOT-tests.jar* のような *jar* ファイルが少なくとも 1 つあります。ファイルの名前は異なる場合がありますが、*-tests.jar* で終わります。

   ```
   .
   |– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |– zip-with-dependencies.zip (this .zip file contains all of the items)
   `– dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
         |– com.some-dependency.bar-4.1.jar
         |– com.another-dependency.thing-1.0.jar
   ```

```
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TES

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テスト JAR ファイル内にクラスファイルが見つかりませんでした。テストパッケージを解凍してから、テスト JAR ファイルをアンジャーし、JAR ファイル内に少なくとも 1 つのクラスファイルがあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   この例の *acme-android-appium-1.0-SNAPSHOT-tests.jar* のような jar ファイルが少なくとも 1 つあります。ファイルの名前は異なる場合がありますが、*-tests.jar* で終わります。

   ```
   .
   |– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |– zip-with-dependencies.zip (this .zip file contains all of the items)
   `– dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
         |– com.some-dependency.bar-4.1.jar
         |– com.another-dependency.thing-1.0.jar
   ```

```
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

3. ファイルを正常に抽出したら、次のコマンドを実行して作業ディレクトリツリーに少なくとも 1 つのクラスがあるはずです:

```
$ tree .
```

次のような出力が表示されます:

```
.
|– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
everything built from the ./src/test directory)
|- one-class-file.class
|- folder
|    `-another-class-file.class
|– zip-with-dependencies.zip (this .zip file contains all of the items)
`– dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_JUNIT_VERSION_VALUE_UNK

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> JUnit のバージョン値が見つかりませんでした。テストパッケージを解凍し、dependency-jars ディレクトリを開き、JUnit JAR ファイルがディレクトリ内にあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip zip-with-dependencies.zip
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して、作業ディレクトリのツリー構造を見つけることができます:

```
tree .
```

出力は次のようになります:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— junit-4.10.jar
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

Appium Java JUnit パッケージが有効な場合は、この例の jar ファイル *junit-4.10.jar* のような JUnit 依存関係ファイルがあります。名前はキーワード *junit* とバージョン番号で構成する必要があります。この例では 4.10 です。

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_INVALID_JUNIT_VERSION

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> JUnit のバージョンが、サポートしている最小バージョン 4.10 よりも低いことがわかりました。JUnit のバージョンを変更して、もう一度試してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip zip-with-dependencies.zip
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

```
$ tree .
```

この例の *junit-4.10.jar* のような JUnit 依存関係ファイルとそのバージョン番号 (この例では 4.10) を見つける必要があります:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— junit-4.10.jar
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

> **Note**
>
> テストパッケージで指定された JUnit のバージョンがサポートしている最小バージョン 4.10 よりも低い場合、テストが正しく実行されないことがあります。

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

AWS Device Farm での Appium Java TestNG テストのトラブルシューティング

以下のトピックでは、Appium Java TestNG テストのアップロード中に発生するエラーメッセージを挙げ、各エラーを解決するための推奨回避策を伝えます。

> **Note**
>
> 以下の手順は Linux x86_64 および Mac を対象にしています。

APPIUM_JAVA_TESTNG_TEST_PACKAGE_UNZIP_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テスト ZIP ファイルを開けませんでした。ファイルが有効であることを確認してから、もう一度お試しください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   有効な Appium Java JUnit パッケージでは、次のような出力が生成されます:

   ```
   .
   |— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   ```

```
|– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|– zip-with-dependencies.zip (this .zip file contains all of the items)
`– dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_JAVA_TESTNG_TEST_PACKAGE_DEPENDENCY_DIR_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> dependency-jars ディレクトリがテストパッケージ内に見つかりませんでした。テストパッケージを解凍し、dependency-jars ディレクトリがパッケージ内にあることを確認して、もう一度試してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Java JUnit パッケージが有効な場合は、作業ディレクトリ内に *dependency-jars* ディレクトリがあります。

   ```
   .
   |– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   ```

```
|– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|– zip-with-dependencies.zip (this .zip file contains all of the items)
`– dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_JAVA_TESTNG_TEST_PACKAGE_JAR_MISSING_IN_DEPENDENCY_

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> dependency-jars ディレクトリツリー内に JAR ファイルが見つかりませんでした。テストパッケージを解凍してから dependency-jars ディレクトリを開き、少なくとも 1 つの JAR ファイルがディレクトリにあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Java JUnit パッケージが有効な場合は、*dependency-jars* ディレクトリ内に少なくとも 1 つの *jar* ファイルがあります。

   ```
   .
   |– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   ```

```
|– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|– zip-with-dependencies.zip (this .zip file contains all of the items)
`– dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_JAVA_TESTNG_TEST_PACKAGE_TESTS_JAR_FILE_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テストパッケージ内に *-tests.jar ファイルが見つかりませんでした。テストパッケージを解凍し、パッケージ内に少なくとも 1 つの *-tests.jar ファイルがあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Java JUnit パッケージが有効な場合は、この例の *acme-android-appium-1.0-SNAPSHOT-tests.jar* のような *jar* ファイルが少なくとも 1 つあります。ファイルの名前は異なる場合がありますが、*-tests.jar* で終わります。

   ```
   .
   ```

```
|– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|– zip-with-dependencies.zip (this .zip file contains all of the items)
`– dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_JAVA_TESTNG_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TESTS

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テスト JAR ファイル内にクラスファイルが見つかりませんでした。テストパッケージを解凍してから、テスト JAR ファイルをアンジャーし、JAR ファイル内に少なくとも 1 つのクラスファイルがあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   この例の *acme-android-appium-1.0-SNAPSHOT-tests.jar* のような jar ファイルが少なくとも 1 つあります。ファイルの名前は異なる場合がありますが、*-tests.jar* で終わります。

```
.
|– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|– zip-with-dependencies.zip (this .zip file contains all of the items)
`– dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

3. その jar ファイルからファイルを抽出するには、次のコマンドを実行します:

```
$ jar xf acme-android-appium-1.0-SNAPSHOT-tests.jar
```

4. ファイルの抽出が正常に完了したら、次のコマンドを実行します:

```
$ tree .
```

作業ディレクトリツリーで、少なくとも 1 つのクラスがあるはずです:

```
.
|– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
everything built from the ./src/test directory)
|- one-class-file.class
|- folder
|    `– another-class-file.class
|– zip-with-dependencies.zip (this .zip file contains all of the items)
`– dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

AWS Device Farm での Appium Java TestNG ウェブアプリケーションのトラブルシューティング

次のトピックでは、Appium Java TestNG ウェブアプリケーションテストのアップロード中に発生するエラーメッセージを挙げ、各エラーを解決するための推奨回避策を伝えます。

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_UNZIP_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テスト ZIP ファイルを開けませんでした。ファイルが有効であることを確認してから、もう一度お試しください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   有効な Appium Java JUnit パッケージでは、次のような出力が生成されます:

   ```
   .
   |— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |— zip-with-dependencies.zip (this .zip file contains all of the items)
   `— dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
         |— com.some-dependency.bar-4.1.jar
         |— com.another-dependency.thing-1.0.jar
   ```

```
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_DEPENDENCY_DIR_MISSIN

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> dependency-jars のディレクトリがテストパッケージ内に見つかりませんでした。テストパッケージを解凍し、dependency-jars ディレクトリがパッケージ内にあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Java JUnit パッケージが有効な場合は、作業ディレクトリ内に *dependency-jars* ディレクトリがあります。

   ```
   .
   |– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |– zip-with-dependencies.zip (this .zip file contains all of the items)
   `– dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
         |– com.some-dependency.bar-4.1.jar
         |– com.another-dependency.thing-1.0.jar
   ```

```
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_JAR_MISSING_IN_DEPENDE

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> dependency-jars ディレクトリツリー内に JAR ファイルが見つかりませんでした。テストパッケージを解凍してから dependency-jars ディレクトリを開き、少なくとも 1 つの JAR ファイルがディレクトリにあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Java JUnit パッケージが有効な場合は、*dependency-jars* ディレクトリ内に少なくとも 1 つの *jar* ファイルがあります。

   ```
   .
   |– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |– zip-with-dependencies.zip (this .zip file contains all of the items)
   `– dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
         |– com.some-dependency.bar-4.1.jar
         |– com.another-dependency.thing-1.0.jar
   ```

```
    |– joda-time-2.7.jar
    `– log4j-1.2.14.jar
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_TESTS_JAR_FILE_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テストパッケージ内に *-tests.jar ファイルが見つかりませんでした。テストパッケージを解凍し、パッケージ内に少なくとも 1 つの *-tests.jar ファイルがあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Java JUnit パッケージが有効な場合は、この例の *acme-android-appium-1.0-SNAPSHOT-tests.jar* のような *jar* ファイルが少なくとも 1 つあります。ファイルの名前は異なる場合がありますが、*-tests.jar* で終わります。

   ```
   .
   |– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |– zip-with-dependencies.zip (this .zip file contains all of the items)
   `– dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
   ```

```
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_CLASS_FILE_MISSING_IN_T

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テスト JAR ファイル内にクラスファイルが見つかりませんでした。テストパッケージを解凍してから、テスト JAR ファイルをアンジャーし、JAR ファイル内に少なくとも 1 つのクラスファイルがあることを確認して、もう一度やり直してください。

次の例では、パッケージ名は「zip-with-dependencies.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   この例の *acme-android-appium-1.0-SNAPSHOT-tests.jar* のような jar ファイルが少なくとも 1 つあります。ファイルの名前は異なる場合がありますが、*-tests.jar* で終わります。

   ```
   .
   |– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |– zip-with-dependencies.zip (this .zip file contains all of the items)
   ```

```
`– dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

3. その jar ファイルからファイルを抽出するには、次のコマンドを実行します:

```
$ jar xf acme-android-appium-1.0-SNAPSHOT-tests.jar
```

4. ファイルの抽出が正常に完了したら、次のコマンドを実行します:

```
$ tree .
```

作業ディレクトリツリーで、少なくとも 1 つのクラスがあるはずです:

```
.
|– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
everything built from the ./src/test directory)
|- one-class-file.class
|- folder
|    `– another-class-file.class
|– zip-with-dependencies.zip (this .zip file contains all of the items)
`– dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

AWS Device Farm での Appium Python テストのトラブルシューティング

次のトピックでは、Appium Python テストのアップロード中に発生するエラーメッセージを挙げ、各エラーを解決するための推奨回避策を伝えます。

APPIUM_PYTHON_TEST_PACKAGE_UNZIP_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> Appium テスト ZIP ファイルを開くことができませんでした。ファイルが有効であることを確認してから、もう一度お試しください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip test_bundle.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   有効な Appium Python パッケージでは、次のような出力が生成されます:

   ```
   .
   |-- requirements.txt
   |-- test_bundle.zip
   |-- tests (directory)
   |       `-- test_unittest.py
   `-- wheelhouse (directory)
           |-- Appium_Python_Client-0.20-cp27-none-any.whl
           |-- py-1.4.31-py2.py3-none-any.whl
           |-- pytest-2.9.0-py2.py3-none-any.whl
           |-- selenium-2.52.0-cp27-none-any.whl
           `-- wheel-0.26.0-py2.py3-none-any.whl
   ```

   詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_PYTHON_TEST_PACKAGE_DEPENDENCY_WHEEL_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> wheelhouse ディレクトリツリーに依存ホイールファイルが見つかりませんでした。テストパッケージを解凍して wheelhouse ディレクトリを開き、少なくとも 1 つのホイールファイルがディレクトリにあることを確認して、もう一度やり直してください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip test_bundle.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Python パッケージが有効な場合、*wheelhouse* ディレクトリ内のハイライトされたファイルのような、*.whl* 依存ファイルが少なくとも 1 つ見つかります。

   ```
   .
   |-- requirements.txt
   |-- test_bundle.zip
   |-- tests (directory)
   |       `-- test_unittest.py
   `-- wheelhouse (directory)
           |-- Appium_Python_Client-0.20-cp27-none-any.whl
           |-- py-1.4.31-py2.py3-none-any.whl
           |-- pytest-2.9.0-py2.py3-none-any.whl
           |-- selenium-2.52.0-cp27-none-any.whl
           `-- wheel-0.26.0-py2.py3-none-any.whl
   ```

   詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_PYTHON_TEST_PACKAGE_INVALID_PLATFORM

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> サポートされていないプラットフォームを指定した wheel ファイルが少なくとも 1 つ見つかりました。テストパッケージを解凍して wheelhouse ディレクトリを開き、ホイールファイルの名前が -any.whl または -linux_x86_64.whl で終わっていることを確認して、もう一度やり直してください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は test_bundle.zip です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip test_bundle.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Python パッケージが有効な場合、*wheelhouse* ディレクトリ内のハイライトされたファイルのような、*.whl* 依存ファイルが少なくとも 1 つ見つかります。ファイルの名前は異なる場合がありますが、*-any.whl* または *-linux_x86_64.whl* で終わる必要があり、これはプラットフォームを指定します。windows のような他のプラットフォームはサポートされていません。

   ```
   .
   |-- requirements.txt
   |-- test_bundle.zip
   |-- tests (directory)
   |       `-- test_unittest.py
   `-- wheelhouse (directory)
           |-- Appium_Python_Client-0.20-cp27-none-any.whl
           |-- py-1.4.31-py2.py3-none-any.whl
           |-- pytest-2.9.0-py2.py3-none-any.whl
           |-- selenium-2.52.0-cp27-none-any.whl
   ```

```
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_PYTHON_TEST_PACKAGE_TEST_DIR_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テストパッケージ内にテストディレクトリが見つかりませんでした。テストパッケージを解凍し、tests ディレクトリがパッケージ内にあることを確認して、もう一度試してください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip test_bundle.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Python パッケージが有効な場合、*tests* ディレクトリは作業ディレクトリ内にあります。

   ```
   .
   |-- requirements.txt
   |-- test_bundle.zip
   |-- tests (directory)
   |       `-- test_unittest.py
   `-- wheelhouse (directory)
           |-- Appium_Python_Client-0.20-cp27-none-any.whl
           |-- py-1.4.31-py2.py3-none-any.whl
           |-- pytest-2.9.0-py2.py3-none-any.whl
           |-- selenium-2.52.0-cp27-none-any.whl
   ```

```
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_PYTHON_TEST_PACKAGE_INVALID_TEST_FILE_NAME

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> tests ディレクトリツリーに有効なテストファイルが見つかりませんでした。テストパッケージを解凍して tests ディレクトリを開き、少なくとも 1 つのファイルの名前が「test」というキーワードで開始または終了していることを確認して、もう一度やり直してください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip test_bundle.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Python パッケージが有効な場合、*tests* ディレクトリは作業ディレクトリ内にあります。ファイルの名前は異なる場合がありますが、*test_* で始まるか、*_test.py* で終わる必要があります。

   ```
   .
   |-- requirements.txt
   |-- test_bundle.zip
   |-- tests (directory)
   |       `-- test_unittest.py
   `-- wheelhouse (directory)
           |-- Appium_Python_Client-0.20-cp27-none-any.whl
           |-- py-1.4.31-py2.py3-none-any.whl
   ```

```
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_PYTHON_TEST_PACKAGE_REQUIREMENTS_TXT_FILE_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テストパッケージ内に requirements.txt ファイルが見つかりませんでした。テストパッケージを解凍し、requirements.txt ファイルがパッケージ内にあることを確認して、もう一度試してください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip test_bundle.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   Appium Python パッケージが有効な場合、*requirements.txt* ファイルは作業ディレクトリ内にあります。

   ```
   .
   |-- requirements.txt
   |-- test_bundle.zip
   |-- tests (directory)
   |       `-- test_unittest.py
   `-- wheelhouse (directory)
           |-- Appium_Python_Client-0.20-cp27-none-any.whl
   ```

```
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_PYTHON_TEST_PACKAGE_INVALID_PYTEST_VERSION

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> サポートされる最低バージョン 2.8.0 より低い pytest バージョンが見つかりました。requirements.txt ファイル内の pytest バージョンを変更して、もう一度試してください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip test_bundle.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   作業ディレクトリの中に *requirement.txt* ファイルがあるはずです。

   ```
   .
   |-- requirements.txt
   |-- test_bundle.zip
   |-- tests (directory)
   |       `--test_unittest.py
   `-- wheelhouse (directory)
           |-- Appium_Python_Client-0.20-cp27-none-any.whl
           |-- py-1.4.31-py2.py3-none-any.whl
   ```

```
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

3. pytest バージョンを取得するには、次のコマンドを実行します:

```
$ grep "pytest" requirements.txt
```

次のような出力があります:

```
pytest==2.9.0
```

pytest バージョンを示しており、この例では 2.9.0 です。Appium Python パッケージが有効な場合、pytest バージョンは 2.8.0 以上である必要があります。

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_PYTHON_TEST_PACKAGE_INSTALL_DEPENDENCY_WHEELS_FAIL

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> 依存ホイールのインストールに失敗しました。テストパッケージを解凍し、requirements.txt ファイルと wheelhouse ディレクトリを開き、requirements.txt ファイルで指定された依存ホイールが wheelhouse ディレクトリ内の依存ホイールと正確に一致することを確認して、もう一度やり直してください。

パッケージングテストのために Python virtualenv をセットアップすることを強くお勧めします。次に、Python virtualenv を使って仮想環境を作成し、それを起動する流れの例を示します:

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip test_bundle.zip
```

2. ホイールファイルのインストールをテストするには、次のコマンドを実行します:

```
$ pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./
requirements.txt
```

有効な Appium Python パッケージでは、次のような出力が生成されます:

```
Ignoring indexes: https://pypi.python.org/simple
Collecting Appium-Python-Client==0.20 (from -r ./requirements.txt (line 1))
Collecting py==1.4.31 (from -r ./requirements.txt (line 2))
Collecting pytest==2.9.0 (from -r ./requirements.txt (line 3))
Collecting selenium==2.52.0 (from -r ./requirements.txt (line 4))
Collecting wheel==0.26.0 (from -r ./requirements.txt (line 5))
Installing collected packages: selenium, Appium-Python-Client, py, pytest, wheel
  Found existing installation: wheel 0.29.0
    Uninstalling wheel-0.29.0:
      Successfully uninstalled wheel-0.29.0
Successfully installed Appium-Python-Client-0.20 py-1.4.31 pytest-2.9.0
 selenium-2.52.0 wheel-0.26.0
```

3. 仮想環境を無効にするには、次のコマンドを実行します:

```
$ deactivate
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_PYTHON_TEST_PACKAGE_PYTEST_COLLECT_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テストディレクトリでテストを収集できませんでした。テストパッケージを解凍して、py.test --collect-only <path to your tests directory> コマンドの実行

が有効であることを確認し、コマンドがエラーを出力しなくなってからもう一度試してください。

パッケージングテストのために Python virtualenv を設定することを強くお勧めします。次に、Python virtualenv を使って仮想環境を作成し、それを起動する流れの例を示します:

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip test_bundle.zip
   ```

2. ホイールファイルをインストールするには、次のコマンドを実行します:

   ```
   $ pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./
   requirements.txt
   ```

3. テストを収集するには、次のコマンドを実行します:

   ```
   $ py.test --collect-only tests
   ```

   有効な Appium Python パッケージでは、次のような出力が生成されます:

   ```
   ==================== test session starts ====================
   platform darwin -- Python 2.7.11, pytest-2.9.0, py-1.4.31, pluggy-0.3.1
   rootdir: /Users/zhena/Desktop/Ios/tests, inifile:
   collected 1 items
   <Module 'test_unittest.py'>
     <UnitTestCase 'DeviceFarmAppiumWebTests'>
       <TestCaseFunction 'test_devicefarm'>

   ==================== no tests ran in 0.11 seconds ====================
   ```

4. 仮想環境を無効にするには、次のコマンドを実行します:

```
$ deactivate
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_PYTHON_TEST_PACKAGE_DEPENDENCY_WHEELS_SUFFICIENT

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> wheelhouse ディレクトリツリーに十分なホイール依存関係が見つかりませんでした。テストパッケージを解凍し、wheelhouse ディレクトリを開いてください。requirements.txt ファイルにホイール依存関係がすべて指定されていることを確認してください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip test_bundle.zip
   ```

2. *requirements.txt* ファイルの長さと、wheelhouse ディレクトリ内の *.whl* 依存ファイルの数を確認します:

   ```
   $ cat requirements.txt | egrep "." |wc -l
       12
   $ ls wheelhouse/ | egrep ".+\.whl" | wc -l
       11
   ```

   .whl 依存ファイルの数が *requirements.txt* ファイル内の空でない行の数よりも少ない場合は、次の点を確かめる必要があります:

   - *requirements.txt* ファイルの各行に対応する *.whl* 依存ファイルがあります。
   - *requirements.txt* ファイルには、依存パッケージ名以外の情報を含む行はありません。
   - *requirements.txt* ファイルでは、依存関係の名前が複数行で重複せず、ファイル内の 2 行が 1 つの *.whl* 依存ファイルに対応している場合があります。

AWS Device Farm は、*requirements.txt* ファイル内の行で依存パッケージに直接対応しないもの (`pip install`コマンドのグローバルオプションを指定する行など) をサポートしていません。グローバルオプションのリストについては、「要件ファイルフォーマット」を参照してください。

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

AWS Device Farm での Appium Python ウェブアプリケーションテストのトラブルシューティング

次のトピックでは、Appium Python ウェブアプリケーションテストのアップロード中に発生するエラーメッセージを挙げ、各エラーを解決するための推奨回避策を示します。

APPIUM_WEB_PYTHON_TEST_PACKAGE_UNZIP_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> Appium テスト ZIP ファイルを開くことができませんでした。ファイルが有効であることを確認してから、もう一度お試しください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip test_bundle.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   有効な Appium Python パッケージでは、次のような出力が生成されます:

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_WEB_PYTHON_TEST_PACKAGE_DEPENDENCY_WHEEL_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> wheelhouse ディレクトリツリーに依存ホイールファイルが見つかりませんでした。テストパッケージを解凍して wheelhouse ディレクトリを開き、少なくとも 1 つのホイールファイルがディレクトリにあることを確認して、もう一度やり直してください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip test_bundle.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

Appium Python パッケージが有効な場合、*wheelhouse* ディレクトリ内のハイライトされたファイルのような、*.whl* 依存ファイルが少なくとも 1 つ見つかります。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_PLATFORM

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> サポートされていないプラットフォームを指定した wheel ファイルが少なくとも 1 つ見つかりました。テストパッケージを解凍して wheelhouse ディレクトリを開き、ホイールファイルの名前が -any.whl または -linux_x86_64.whl で終わっていることを確認して、もう一度やり直してください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip test_bundle.zip
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

```
$ tree .
```

Appium Python パッケージが有効な場合、*wheelhouse* ディレクトリ内のハイライトされたファイルのような、*.whl* 依存ファイルが少なくとも 1 つ見つかります。ファイルの名前は異なる場合がありますが、*-any.whl* または *-linux_x86_64.whl* で終わる必要があり、これはプラットフォームを指定します。windows のような他のプラットフォームはサポートされていません。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_WEB_PYTHON_TEST_PACKAGE_TEST_DIR_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テストパッケージ内にテストディレクトリが見つかりませんでした。テストパッケージを解凍し、tests ディレクトリがパッケージ内にあることを確認して、もう一度試してください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip test_bundle.zip
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

```
$ tree .
```

Appium Python パッケージが有効な場合、*tests* ディレクトリは作業ディレクトリ内にあります。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_TEST_FILE_NAME

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> tests ディレクトリツリーに有効なテストファイルが見つかりませんでした。テストパッケージを解凍して tests ディレクトリを開き、少なくとも 1 つのファイルの名前が「test」というキーワードで開始または終了していることを確認して、もう一度やり直してください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip test_bundle.zip
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

```
$ tree .
```

Appium Python パッケージが有効な場合、*tests* ディレクトリは作業ディレクトリ内にあります。ファイルの名前は異なる場合がありますが、*test_* で始まるか、*_test.py* で終わる必要があります。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_WEB_PYTHON_TEST_PACKAGE_REQUIREMENTS_TXT_FILE_MISS

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テストパッケージ内に requirements.txt ファイルが見つかりませんでした。テストパッケージを解凍し、requirements.txt ファイルがパッケージ内にあることを確認して、もう一度試してください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip test_bundle.zip
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

```
$ tree .
```

Appium Python パッケージが有効な場合、*requirements.txt* ファイルは作業ディレクトリ内にあります。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_PYTEST_VERSION

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> サポートされる最低バージョン 2.8.0 より低い pytest バージョンが見つかりました。requirements.txt ファイル内の pytest バージョンを変更して、もう一度試してください。

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip test_bundle.zip
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

```
$ tree .
```

作業ディレクトリの中に *requirement.txt* ファイルがあるはずです。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `--test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

3. pytest のバージョンを取得するには、次のコマンドを実行します:

```
$ grep "pytest" requirements.txt
```

次のような出力があります:

```
pytest==2.9.0
```

pytest バージョンを示しており、この例では 2.9.0 です。Appium Python パッケージが有効な場合、pytest バージョンは 2.8.0 以上である必要があります。

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_WEB_PYTHON_TEST_PACKAGE_INSTALL_DEPENDENCY_WHEELS

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> 依存ホイールのインストールに失敗しました。テストパッケージを解凍し、requirements.txt ファイルと wheelhouse ディレクトリを開き、requirements.txt ファイルで指定された依存ホイールが wheelhouse ディレクトリ内の依存ホイールと正確に一致することを確認して、もう一度やり直してください。

パッケージングテストのために Python virtualenv を設定することを強くお勧めします。次に、Python virtualenv を使って仮想環境を作成し、それを起動する流れの例を示します:

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

   ```
   $ unzip test_bundle.zip
   ```

2. ホイールファイルのインストールをテストするには、次のコマンドを実行します:

   ```
   $ pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./
   requirements.txt
   ```

   有効な Appium Python パッケージでは、次のような出力が生成されます:

   ```
   Ignoring indexes: https://pypi.python.org/simple
   Collecting Appium-Python-Client==0.20 (from -r ./requirements.txt (line 1))
   Collecting py==1.4.31 (from -r ./requirements.txt (line 2))
   Collecting pytest==2.9.0 (from -r ./requirements.txt (line 3))
   Collecting selenium==2.52.0 (from -r ./requirements.txt (line 4))
   Collecting wheel==0.26.0 (from -r ./requirements.txt (line 5))
   Installing collected packages: selenium, Appium-Python-Client, py, pytest, wheel
     Found existing installation: wheel 0.29.0
       Uninstalling wheel-0.29.0:
         Successfully uninstalled wheel-0.29.0
   ```

```
Successfully installed Appium-Python-Client-0.20 py-1.4.31 pytest-2.9.0
 selenium-2.52.0 wheel-0.26.0
```

3. 仮想環境を無効にするには、次のコマンドを実行します:

```
$ deactivate
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

APPIUM_WEB_PYTHON_TEST_PACKAGE_PYTEST_COLLECT_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テストディレクトリでテストを収集できませんでした。テストパッケージを解凍し、"py.test --collect-only <path to your tests directory>" コマンドを実行してテストパッケージが有効であることを確認します。コマンドでエラーが出力されなくなったら再度実行してください。

パッケージングテストのために Python virtualenv を設定することを強くお勧めします。次に、Python virtualenv を使って仮想環境を作成し、それを起動する流れの例を示します:

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

エラーなしでテストパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「test_bundle.zip」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip test_bundle.zip
```

2. ホイールファイルをインストールするには、次のコマンドを実行します:

```
$ pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./
requirements.txt
```

3. テストを収集するには、次のコマンドを実行します:

```
$ py.test --collect-only tests
```

有効な Appium Python パッケージでは、次のような出力が生成されます:

```
=================== test session starts ====================
platform darwin -- Python 2.7.11, pytest-2.9.0, py-1.4.31, pluggy-0.3.1
rootdir: /Users/zhena/Desktop/Ios/tests, inifile:
collected 1 items
<Module 'test_unittest.py'>
  <UnitTestCase 'DeviceFarmAppiumWebTests'>
    <TestCaseFunction 'test_devicefarm'>

=================== no tests ran in 0.11 seconds ====================
```

4. 仮想環境を無効にするには、次のコマンドを実行します:

```
$ deactivate
```

詳細については、「Appium と AWS Device Farm による作業」を参照してください。

AWS Device Farm でのインストルメンテーションテストのトラブルシューティング

次のトピックでは、インストルメンテーションテストのアップロード中に発生するエラーメッセージを挙げ、各エラーを解決するための推奨回避策を示します。

INSTRUMENTATION_TEST_PACKAGE_UNZIP_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テスト APK ファイルを開けませんでした。ファイルが有効であることを確認してから、もう一度お試しください。

エラーなしでテストパッケージを解凍できることを確かめます。次の例では、パッケージ名は「app-debug-androidTest-unaligned.apk」です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ unzip app-debug-androidTest-unaligned.apk
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

```
$ tree .
```

有効なインストルメンテーションテストパッケージでは、次のような出力が生成されます:

```
.
|-- AndroidManifest.xml
|--  classes.dex
|-- resources.arsc
|-- LICENSE-junit.txt
|-- junit (directory)
`-- META-INF (directory)
```

詳細については、「Android および AWS Device Farm のインストゥルメンテーションによる作業」を参照してください。

INSTRUMENTATION_TEST_PACKAGE_AAPT_DEBUG_BADGING_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> テストパッケージに関する情報を抽出できませんでした。"aapt debug badging <path to your test package>" コマンドを実行してテストパッケージが有効であることを確認します。コマンドでエラーが出力されなくなったら一度試してください。

アップロード検証プロセス中に、Device Farm は `aapt debug badging <path to your package>` コマンドの出力から情報を解析します。

インストルメンテーションテストパッケージでこのコマンドを正常に実行できることを確かめます。

次の例では、パッケージ名は「app-debug-androidTest-unaligned.apk」です。

- テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ aapt debug badging app-debug-androidTest-unaligned.apk
```

有効なインストルメンテーションテストパッケージでは、次のような出力が生成されます:

```
package: name='com.amazon.aws.adf.android.referenceapp.test' versionCode=''
 versionName='' platformBuildVersionName='5.1.1-1819727'
sdkVersion:'9'
targetSdkVersion:'22'
application-label:'Test-api'
application: label='Test-api' icon=''
application-debuggable
uses-library:'android.test.runner'
feature-group: label=''
uses-feature: name='android.hardware.touchscreen'
uses-implied-feature: name='android.hardware.touchscreen' reason='default feature
 for all apps'
supports-screens: 'small' 'normal' 'large' 'xlarge'
supports-any-density: 'true'
locales: '--_--'
densities: '160'
```

詳細については、「Android および AWS Device Farm のインストゥルメンテーションによる作業」を参照してください。

INSTRUMENTATION_TEST_PACKAGE_INSTRUMENTATION_RUNNER_VALU

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> AndroidManifest.xml に、インストルメンテーションランナーの値が見つかりませんでした。"aapt dump xmltree <path to your test package> AndroidManifest.xml" コマンドを実行してテストパッケージが有効であることを確認し、キーワード "instrumentation" の後ろにインストルメンテーションランナーの値を見つけた後にもう一度試して下さい。

アップロード検証プロセス中に、Device Farm はパッケージに含まれる XML ファイルの XML 解析ツリーからインストルメンテーションランナーの値を解析します。以下の `aapt dump xmltree <path to your package> AndroidManifest.xml` コマンドを使用できます:

インストルメンテーションテストパッケージでこのコマンドを実行でき、インストルメンテーションの値を正常に見つけられることを確かめます。

次の例では、パッケージ名は「app-debug-androidTest-unaligned.apk」です。

- テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ aapt dump xmltree app-debug-androidTest-unaligned.apk AndroidManifest.xml | grep
 -A5 "instrumentation"
```

有効なインストルメンテーションテストパッケージでは、次のような出力が生成されます:

```
E: instrumentation (line=9)
      A: android:label(0x01010001)="Tests for
 com.amazon.aws.adf.android.referenceapp" (Raw: "Tests for
 com.amazon.aws.adf.android.referenceapp")
      A:
 android:name(0x01010003)="android.support.test.runner.AndroidJUnitRunner" (Raw:
 "android.support.test.runner.AndroidJUnitRunner")
      A:
 android:targetPackage(0x01010021)="com.amazon.aws.adf.android.referenceapp" (Raw:
 "com.amazon.aws.adf.android.referenceapp")
      A: android:handleProfiling(0x01010022)=(type 0x12)0x0
      A: android:functionalTest(0x01010023)=(type 0x12)0x0
```

詳細については、「Android および AWS Device Farm のインストゥルメンテーションによる作業」を参照してください。

INSTRUMENTATION_TEST_PACKAGE_AAPT_DUMP_XMLTREE_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> 有効な AndroidManifest.xml がテストパッケージ内に見つかりませんでした。"aapt dump xmltree <path to your test package> AndroidManifest.xml" コマンドを実行してテストパッ

ケージが有効であることを確認します。コマンドでエラーが出力されくなったらもう一度試してください。

アップロード検証プロセス中に、Device Farm は `aapt dump xmltree <path to your package> AndroidManifest.xml` コマンドを使用してパッケージに含まれる XML ファイルの XML 解析ツリーから情報を解析します:

インストルメンテーションテストパッケージでこのコマンドを正常に実行できることを確かめます。

次の例では、パッケージ名は「app-debug-androidTest-unaligned.apk」です。

- テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

```
$ aapt dump xmltree app-debug-androidTest-unaligned.apk AndroidManifest.xml
```

有効なインストルメンテーションテストパッケージでは、次のような出力が生成されます:

```
N: android=http://schemas.android.com/apk/res/android
  E: manifest (line=2)
    A: package="com.amazon.aws.adf.android.referenceapp.test" (Raw:
 "com.amazon.aws.adf.android.referenceapp.test")
    A: platformBuildVersionCode=(type 0x10)0x16 (Raw: "22")
    A: platformBuildVersionName="5.1.1-1819727" (Raw: "5.1.1-1819727")
    E: uses-sdk (line=5)
      A: android:minSdkVersion(0x0101020c)=(type 0x10)0x9
      A: android:targetSdkVersion(0x01010270)=(type 0x10)0x16
    E: instrumentation (line=9)
      A: android:label(0x01010001)="Tests for
 com.amazon.aws.adf.android.referenceapp" (Raw: "Tests for
 com.amazon.aws.adf.android.referenceapp")
      A:
 android:name(0x01010003)="android.support.test.runner.AndroidJUnitRunner" (Raw:
 "android.support.test.runner.AndroidJUnitRunner")
      A:
 android:targetPackage(0x01010021)="com.amazon.aws.adf.android.referenceapp" (Raw:
 "com.amazon.aws.adf.android.referenceapp")
      A: android:handleProfiling(0x01010022)=(type 0x12)0x0
      A: android:functionalTest(0x01010023)=(type 0x12)0x0
    E: application (line=16)
      A: android:label(0x01010001)=@0x7f020000
      A: android:debuggable(0x0101000f)=(type 0x12)0xffffffff
```

```
      E: uses-library (line=17)
        A: android:name(0x01010003)="android.test.runner" (Raw:
 "android.test.runner")
```

詳細については、「Android および AWS Device Farm のインストゥルメンテーションによる作業」を参照してください。

INSTRUMENTATION_TEST_PACKAGE_TEST_PACKAGE_NAME_VALUE_MISS

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
> テストパッケージ内にパッケージ名が見つかりませんでした。"aapt debug badging <path to your test package>" コマンドを実行してテストパッケージが有効であることを確認し、キーワード "package: name" の後ろにパッケージ名の値を見つけた後にもう一度試して下さい。

アップロード検証プロセス中に、Device Farm は `aapt debug badging <path to your package>` コマンドの出力からパッケージ名の値を解析します:

インストルメンテーションテストパッケージでこのコマンドを実行でき、パッケージ名の値を正常に見つけられることを確かめます。

次の例では、パッケージ名は「app-debug-androidTest-unaligned.apk」です。

- テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します:

  ```
  $ aapt debug badging app-debug-androidTest-unaligned.apk | grep "package: name="
  ```

  有効なインストルメンテーションテストパッケージでは、次のような出力が生成されます:

  ```
  package: name='com.amazon.aws.adf.android.referenceapp.test' versionCode=''
   versionName='' platformBuildVersionName='5.1.1-1819727'
  ```

  詳細については、「Android および AWS Device Farm のインストゥルメンテーションによる作業」を参照してください。

AWS Device Farm の iOS アプリケーションテストのトラブルシューティング

次のトピックでは、iOS アプリケーションテストのアップロード中に発生するエラーメッセージを挙げ、各エラーを解決するための推奨回避策を示します。

> **Note**
>
> 以下の手順は Linux x86_64 および Mac を対象にしています。

IOS_APP_UNZIP_FAILED

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> アプリケーションを開けませんでした。ファイルが有効であることを確認してから、もう一度お試しください。

エラーなしでアプリケーションパッケージを解凍できることを確かめてください。次の例では、パッケージの名前は「AWSDeviceFarmiOSReferenceApp.ipa」です。

1. アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   有効な iOS アプリケーションパッケージでは、次のような出力が生成されます:

   ```
   .
   `-- Payload (directory)
           `-- AWSDeviceFarmiOSReferenceApp.app (directory)
   ```

```
                    |-- Info.plist
                    `-- (any other files)
```

詳細については、「AWS Device Farm での iOS テストによる作業。」を参照してください。

IOS_APP_PAYLOAD_DIR_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> アプリケーション内に Payload ディレクトリが見つかりませんでした。アプリケーションを解凍し、Payload ディレクトリがパッケージ内にあることを確認して、もう一度試してください。

次の例では、パッケージの名前は「AWSDeviceFarmiOSReferenceApp.ipa」です。

1. アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

```
$ tree .
```

iOS アプリケーションパッケージが有効な場合、*Payload* ディレクトリは作業ディレクトリ内にあります。

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                    |-- Info.plist
                    `-- (any other files)
```

詳細については、「AWS Device Farm での iOS テストによる作業。」を参照してください。

IOS_APP_APP_DIR_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> Payload ディレクトリ内に .app ディレクトリが見つかりませんでした。アプリケーションを解凍し、次に Payload ディレクトリを開き .app ディレクトリがディレクトリ内にあることを確認して、もう一度試してください。

次の例では、パッケージの名前は「AWSDeviceFarmiOSReferenceApp.ipa」です。

1. アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   iOS アプリケーションパッケージが有効な場合、*Payload* ディレクトリ内にこの例の *AWSDeviceFarmiOSReferenceApp.app* のような *.app* ディレクトリがあります。

   ```
   .
   `-- Payload (directory)
           `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                       |-- Info.plist
                       `-- (any other files)
   ```

   詳細については、「AWS Device Farm での iOS テストによる作業。」を参照してください。

IOS_APP_PLIST_FILE_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> .app ディレクトリ内に Info.plist ファイルが見つかりませんでした。アプリケーションを解凍し、次に .app ディレクトリを開き Info.plist ファイルがディレクトリ内にあることを確認して、もう一度試してください。

次の例では、パッケージの名前は「AWSDeviceFarmiOSReferenceApp.ipa」です。

1. アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   iOS アプリケーションパッケージが有効な場合、この例の *AWSDeviceFarmiOSReferenceApp.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあります。

   ```
   .
   `-- Payload (directory)
           `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                       |-- Info.plist
                       `-- (any other files)
   ```

   詳細については、「AWS Device Farm での iOS テストによる作業。」を参照してください。

IOS_APP_CPU_ARCHITECTURE_VALUE_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist ファイルに CPU アーキテクチャの値が見つかりませんでした。アプリケーションを解凍し、次に .app ディレクトリ内の Info.plist ファイルを開き、キー "UIRequiredDeviceCapabilities" が指定されていることを確認して、もう一度試してください。

次の例では、パッケージの名前は「AWSDeviceFarmiOSReferenceApp.ipa」です。

1. アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   この例の *AWSDeviceFarmiOSReferenceApp.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあるはずです:

   ```
   .
   `-- Payload (directory)
           `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                       |-- Info.plist
                       `-- (any other files)
   ```

3. CPU アーキテクチャの値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

   Python の場合、次のコマンドを実行して biplist モジュールをインストールできます:

   ```
   $ pip install biplist
   ```

4. 次に、Python を開き、次のコマンドを入力します:

   ```
   import biplist
   ```

```
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
Info.plist')
print info_plist['UIRequiredDeviceCapabilities']
```

有効な iOS アプリケーションパッケージでは、次のような出力が生成されます:

```
['armv7']
```

詳細については、「AWS Device Farm での iOS テストによる作業。」を参照してください。

IOS_APP_PLATFORM_VALUE_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist ファイルにプラットフォームの値が見つかりませんでした。アプリケーションを解凍し、次に .app ディレクトリ内の Info.plist ファイルを開き、キー "CFBundleSupportedPlatforms" が指定されていることを確認して、もう一度試してください。

次の例では、パッケージの名前は「AWSDeviceFarmiOSReferenceApp.ipa」です。

1. アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   この例の *AWSDeviceFarmiOSReferenceApp.app* のような *.app* ディレクトリ内に Info.plist ファイルがあるはずです:

   ```
   .
   ```

```
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                        |-- Info.plist
                        `-- (any other files)
```

3. プラットフォームの値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

   Python の場合、次のコマンドを実行して biplist モジュールをインストールできます:

   ```
   $ pip install biplist
   ```

4. 次に、Python を開き、次のコマンドを入力します:

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
   Info.plist')
   print info_plist['CFBundleSupportedPlatforms']
   ```

   有効な iOS アプリケーションパッケージでは、次のような出力が生成されます:

   ```
   ['iPhoneOS']
   ```

   詳細については、「AWS Device Farm での iOS テストによる作業。」を参照してください。

IOS_APP_WRONG_PLATFORM_DEVICE_VALUE

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist ファイルでプラットフォームデバイスの値が間違っていることが分かりました。アプリケーションを解凍し、次に .app ディレクトリ内の Info.plist ファイルを開き、キーの値 "CFBundleSupportedPlatforms" にキーワード "simulator" が含まれていないことを確認して、もう一度試してください。

次の例では、パッケージの名前は「AWSDeviceFarmiOSReferenceApp.ipa」です。

1. アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

```
$ tree .
```

この例の *AWSDeviceFarmiOSReferenceApp.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあるはずです:

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                        |-- Info.plist
                        `-- (any other files)
```

3. プラットフォームの値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

Python の場合、次のコマンドを実行して biplist モジュールをインストールできます:

```
$ pip install biplist
```

4. 次に、Python を開き、次のコマンドを入力します:

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
Info.plist')
print info_plist['CFBundleSupportedPlatforms']
```

有効な iOS アプリケーションパッケージでは、次のような出力が生成されます:

```
['iPhoneOS']
```

iOS アプリケーションが有効な場合、値にキーワード `simulator` を含めることはできません。

詳細については、「AWS Device Farm での iOS テストによる作業。」を参照してください。

IOS_APP_FORM_FACTOR_VALUE_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist ファイルにフォームファクタの値が見つかりませんでした。アプリケーションを解凍し、次に .app ディレクトリ内の Info.plist ファイルを開き、キー "UIDeviceFamily" が指定されていることを確認して、もう一度試してください。

次の例では、パッケージの名前は「AWSDeviceFarmiOSReferenceApp.ipa」です。

1. アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   この例の *AWSDeviceFarmiOSReferenceApp.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあるはずです:

   ```
   .
   `-- Payload (directory)
           `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                       |-- Info.plist
                       `-- (any other files)
   ```

3. フォームファクタの値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

   Python の場合、次のコマンドを実行して biplist モジュールをインストールできます:

   ```
   $ pip install biplist
   ```

4. 次に、Python を開き、次のコマンドを入力します:

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
Info.plist')
print info_plist['UIDeviceFamily']
```

有効な iOS アプリケーションパッケージでは、次のような出力が生成されます:

```
[1, 2]
```

詳細については、「AWS Device Farm での iOS テストによる作業。」を参照してください。

IOS_APP_PACKAGE_NAME_VALUE_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist ファイルにパッケージ名の値が見つかりませんでした。アプリケーションを解凍し、次に .app ディレクトリ内の Info.plist ファイルを開き、キー "CFBundleIdentifier" が指定されていることを確認して、もう一度試してください。

次の例では、パッケージの名前は「AWSDeviceFarmiOSReferenceApp.ipa」です。

1. アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

   ```
   $ tree .
   ```

   この例の *AWSDeviceFarmiOSReferenceApp.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあるはずです:

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                        |-- Info.plist
                        `-- (any other files)
```

3. パッケージ名の値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

   Python の場合、次のコマンドを実行して biplist モジュールをインストールできます:

```
$ pip install biplist
```

4. 次に、Python を開き、次のコマンドを入力します:

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
Info.plist')
print info_plist['CFBundleIdentifier']
```

   有効な iOS アプリケーションパッケージでは、次のような出力が生成されます:

```
Amazon.AWSDeviceFarmiOSReferenceApp
```

   詳細については、「AWS Device Farm での iOS テストによる作業。」を参照してください。

IOS_APP_EXECUTABLE_VALUE_MISSING

次のメッセージが表示された場合は、下の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist ファイルに実行可能な値が見つかりませんでした。アプリケーションを解凍し、次に .app ディレクトリ内の Info.plist ファイルを開き、キー "CFBundleExecutable" が指定されていることを確認して、もう一度試してください。

次の例では、パッケージの名前は「AWSDeviceFarmiOSReferenceApp.ipa」です。

1. アプリケーションパッケージを作業ディレクトリにコピーし、次に以下のコマンドを実行します:

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます:

```
$ tree .
```

この例の *AWSDeviceFarmiOSReferenceApp.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあるはずです:

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                        |-- Info.plist
                        `-- (any other files)
```

3. 実行可能な値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

Python の場合、次のコマンドを実行して biplist モジュールをインストールできます:

```
$ pip install biplist
```

4. 次に、Python を開き、次のコマンドを入力します:

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
Info.plist')
print info_plist['CFBundleExecutable']
```

有効な iOS アプリケーションパッケージでは、次のような出力が生成されます:

```
AWSDeviceFarmiOSReferenceApp
```

詳細については、「AWS Device Farm での iOS テストによる作業。」を参照してください。

AWS Device Farm での XCTest テストのトラブルシューティング

次のトピックでは、XCTest テストのアップロード中に発生するエラーメッセージを示し、各エラーを解決するための回避策を推奨します。

> **Note**
>
> 以下の指示は MacOS を使用していることを前提としています。

XCTEST_TEST_PACKAGE_UNZIP_FAILED

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> テスト ZIP ファイルを開けませんでした。ファイルが有効であることを確認してから、もう一度お試しください。

エラーなしでアプリケーションパッケージを解凍できることを確認します。次の例では、パッケージ名は [swiftExampleTests.xctest-1.zip] です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

   ```
   $ unzip swiftExampleTests.xctest-1.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

   ```
   $ tree .
   ```

   有効な XCTest パッケージでは、次のような出力が生成されます。

   ```
   .
   `-- swiftExampleTests.xctest (directory)
                |-- Info.plist
                `-- (any other files)
   ```

詳細については、「iOS 用 XCTest と AWS Device Farm による作業」を参照してください。

XCTEST_TEST_PACKAGE_XCTEST_DIR_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> テストパッケージ内に .xctest ディレクトリが見つかりませんでした。テストパッケージを解凍し、.xctest ディレクトリがパッケージ内にあることを確認して、もう一度試してください。

次の例では、パッケージ名は [swiftExampleTests.xctest-1.zip] です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

   ```
   $ unzip swiftExampleTests.xctest-1.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

   ```
   $ tree .
   ```

   XCTest パッケージが有効である場合、作業ディレクトリ内に *swiftExampleTests.xctest* に類似した名前のディレクトリがあります。ディレクトリ名の末尾は *.xctest* です。

   ```
   .
   `-- swiftExampleTests.xctest (directory)
               |-- Info.plist
               `-- (any other files)
   ```

   詳細については、「iOS 用 XCTest と AWS Device Farm による作業」を参照してください。

XCTEST_TEST_PACKAGE_PLIST_FILE_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> .xctest ディレクトリ内に Info.plist ファイルが見つかりませんでした。テストパッケージを解凍し、次に .xctest ディレクトリを開き Info.plist ファイルがディレクトリ内にあることを確認して、もう一度試してください。

次の例では、パッケージ名は [swiftExampleTests.xctest-1.zip] です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

```
$ unzip swiftExampleTests.xctest-1.zip
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

```
$ tree .
```

XCTest パッケージが有効な場合、*Info.plist* ファイルは *.xctest* ディレクトリ内にあります。以下の例では、ディレクトリ名は *swiftExampleTests.xctest* です。

```
.
`-- swiftExampleTests.xctest (directory)
              |-- Info.plist
              `-- (any other files)
```

詳細については、「iOS 用 XCTest と AWS Device Farm による作業」を参照してください。

XCTEST_TEST_PACKAGE_PACKAGE_NAME_VALUE_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist ファイルにパッケージ名の値が見つかりませんでした。テストパッケージを解凍し、次に Info.plist ファイルを開き、「CFBundleIdentifier」というキーが指定されていることを確認して、もう一度試してください。

次の例では、パッケージ名は [swiftExampleTests.xctest-1.zip] です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

```
$ unzip swiftExampleTests.xctest-1.zip
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

```
$ tree .
```

この例の *swiftExampleTests.xctest* のような *.xctest* ディレクトリ内に *Info.plist* ファイルがあるはずです。

```
.
`-- swiftExampleTests.xctest (directory)
              |-- Info.plist
              `-- (any other files)
```

3. パッケージ名の値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

Python の場合、次のコマンドを実行して biplist モジュールをインストールできます。

```
$ pip install biplist
```

4. 次に、Python を開き、次のコマンドを入力します。

```
import biplist
info_plist = biplist.readPlist('swiftExampleTests.xctest/Info.plist')
print info_plist['CFBundleIdentifier']
```

有効な XCtest アプリケーションパッケージでは、次のような出力が生成されます。

```
com.amazon.kanapka.swiftExampleTests
```

詳細については、「iOS 用 XCTest と AWS Device Farm による作業」を参照してください。

XCTEST_TEST_PACKAGE_EXECUTABLE_VALUE_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist ファイルに実行可能な値が見つかりませんでした。テストパッケージを解凍し、次に Info.plist ファイルを開き、「CFBundleExecutable」というキーが指定されていることを確認して、もう一度試してください。

次の例では、パッケージ名は [swiftExampleTests.xctest-1.zip] です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

   ```
   $ unzip swiftExampleTests.xctest-1.zip
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

   ```
   $ tree .
   ```

   この例の *swiftExampleTests.xctest* のような *.xctest* ディレクトリ内に *Info.plist* ファイルがあるはずです。

   ```
   .
   `-- swiftExampleTests.xctest (directory)
                |-- Info.plist
                `-- (any other files)
   ```

3. パッケージ名の値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

   Python の場合、次のコマンドを実行して biplist モジュールをインストールできます。

   ```
   $ pip install biplist
   ```

4. 次に、Python を開き、次のコマンドを入力します。

   ```
   import biplist
   ```

```
info_plist = biplist.readPlist('swiftExampleTests.xctest/Info.plist')
print info_plist['CFBundleExecutable']
```

有効な XCtest アプリケーションパッケージでは、次のような出力が生成されます。

```
swiftExampleTests
```

詳細については、「iOS 用 XCTest と AWS Device Farm による作業」を参照してください。

AWS Device Farm での XCTest UI テストのトラブルシューティング

次のトピックでは、XCTest UI テストのアップロード中に発生するエラーメッセージを示し、各エラーを解決するために推奨される回避策を示します。

> **Note**
>
> 以下の手順は Linux x86_64 および Mac を対象にしています。

XCTEST_UI_TEST_PACKAGE_UNZIP_FAILED

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> テスト IPA ファイルを開けませんでした。ファイルが有効であることを確認してから、もう一度お試しください。

エラーなしでアプリケーションパッケージを解凍できることを確認します。次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

```
$ unzip swift-sample-UI.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

```
$ tree .
```

有効な iOS アプリケーションパッケージでは、次のような出力が生成されます。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_PAYLOAD_DIR_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> Payload のディレクトリがテストパッケージ内に見つかりませんでした。テストパッケージを解凍し、Payload ディレクトリがパッケージ内にあることを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

```
$ unzip swift-sample-UI.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

```
$ tree .
```

XCTest UI パッケージが有効な場合、*Payload* ディレクトリは作業ディレクトリ内にあります。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_APP_DIR_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> Payload ディレクトリ内に .app ディレクトリが見つかりませんでした。テストパッケージを解凍し、次に Payload ディレクトリを開き .app ディレクトリがディレクトリ内にあることを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

   ```
   $ tree .
   ```

XCTest UI パッケージが有効な場合、*Payload* ディレクトリ内にこの例の *swift-sampleUITests-Runner.app* のような *.app* ディレクトリがあります。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                    |-- Info.plist
                    |-- Plugins (directory)
                    |       `swift-sampleUITests.xctest (directory)
                    |                       |-- Info.plist
                    |                       `-- (any other files)
                    `-- (any other files)
```

詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_PLUGINS_DIR_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> .app ディレクトリ内に Plugins ディレクトリが見つかりませんでした。テストパッケージを解凍し、次に .app ディレクトリを開き Plugins ディレクトリがディレクトリ内にあることを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

```
$ unzip swift-sample-UI.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

```
$ tree .
```

XCTest UI パッケージが有効な場合、*Plugins* ディレクトリは *.app* ディレクトリ内にあります。この例では、ディレクトリ名は *swift-sampleUITests-Runner.app* です。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_XCTEST_DIR_MISSING_IN_PLUGINS_DIR

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> Plugins ディレクトリ内に .xctest ディレクトリが見つかりませんでした。テストパッケージを解凍し、次に Plugins ディレクトリを開き .xctest ディレクトリがディレクトリ内にあることを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

```
$ unzip swift-sample-UI.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

```
$ tree .
```

XCTest UI パッケージが有効な場合、*.xctest* ディレクトリは *Plugins* ディレクトリ内にあります。この例では、ディレクトリ名は *swift-sampleUITests.xctest* です。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_PLIST_FILE_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> .app ディレクトリ内に Info.plist ファイルが見つかりませんでした。テストパッケージを解凍し、次に .app ディレクトリを開き Info.plist ファイルがディレクトリ内にあることを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

```
$ unzip swift-sample-UI.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

```
$ tree .
```

XCTest UI パッケージが有効な場合、*Info.plist* ファイルは *.app* ディレクトリ内にあります。次の例では、ディレクトリ名は *swift-sampleUITests-Runner.app* です。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_PLIST_FILE_MISSING_IN_XCTEST_DIR

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> .xctest ディレクトリ内に Info.plist ファイルが見つかりませんでした。テストパッケージを解凍し、次に .xctest ディレクトリを開き Info.plist ファイルがディレクトリ内にあることを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

```
$ unzip swift-sample-UI.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

```
$ tree .
```

XCTest UI アプリケーションパッケージが有効な場合、*Info.plist* ファイルは *.xctest* ディレクトリ内にあります。以下の例では、ディレクトリ名は *swift-sampleUITests.xctest* です。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_CPU_ARCHITECTURE_VALUE_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist ファイルに CPU アーキテクチャの値が見つかりませんでした。テストパッケージを解凍し、次に .app ディレクトリ内の Info.plist ファイルを開き、キー "UIRequiredDeviceCapabilities" が指定されていることを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

   ```
   $ tree .
   ```

この例の *swift-sampleUITests-Runner.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあるはずです。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

3. CPU アーキテクチャの値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

   Python の場合、次のコマンドを実行して biplist モジュールをインストールできます。

   ```
   $ pip install biplist
   ```

4. 次に、Python を開き、次のコマンドを入力します。

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/
   Info.plist')
   print info_plist['UIRequiredDeviceCapabilities']
   ```

   有効な XCtest UI パッケージでは、次のような出力が生成されます。

   ```
   ['armv7']
   ```

   詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_PLATFORM_VALUE_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist にプラットフォームの値が見つかりませんでした。テストパッケージを解凍し、次に .app ディレクトリ内の Info.plist ファイルを開き、キー "CFBundleSupportedPlatforms" が指定されていることを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

```
$ unzip swift-sample-UI.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

```
$ tree .
```

この例の *swift-sampleUITests-Runner.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあるはずです。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                    |-- Info.plist
                    |-- Plugins (directory)
                    |       `swift-sampleUITests.xctest (directory)
                    |                       |-- Info.plist
                    |                       `-- (any other files)
                    `-- (any other files)
```

3. プラットフォームの値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

Python の場合、次のコマンドを実行して biplist モジュールをインストールできます。

```
$ pip install biplist
```

4. 次に、Python を開き、次のコマンドを入力します。

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['CFBundleSupportedPlatforms']
```

有効な XCtest UI パッケージでは、次のような出力が生成されます。

```
['iPhoneOS']
```

詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_WRONG_PLATFORM_DEVICE_VALUE

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist ファイルでプラットフォームデバイスの値が間違っていることが分かりました。テストパッケージを解凍し、次に .app ディレクトリ内の Info.plist ファイルを開き、キーの値 "CFBundleSupportedPlatforms" にキーワード "simulator" が含まれていないことを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

   ```
   $ tree .
   ```

   この例の *swift-sampleUITests-Runner.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあるはずです。

   ```
   .
   ```

```
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

3. プラットフォームの値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

   Python の場合、次のコマンドを実行して biplist モジュールをインストールできます。

   ```
   $ pip install biplist
   ```

4. 次に、Python を開き、次のコマンドを入力します。

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
   print info_plist['CFBundleSupportedPlatforms']
   ```

   有効な XCtest UI パッケージでは、次のような出力が生成されます。

   ```
   ['iPhoneOS']
   ```

   XCTest UI パッケージが有効な場合、値にキーワード `simulator` を含めることはできません。

   詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_FORM_FACTOR_VALUE_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist にフォームファクタの値が見つかりませんでした。テストパッケージを解凍し、次に .app ディレクトリ内の Info.plist ファイルを開き、キー "UIDeviceFamily" が指定されていることを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

```
$ unzip swift-sample-UI.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

```
$ tree .
```

この例の *swift-sampleUITests-Runner.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあるはずです。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

3. フォームファクタの値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

Python の場合、次のコマンドを実行して biplist モジュールをインストールできます。

```
$ pip install biplist
```

4. 次に、Python を開き、次のコマンドを入力します。

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['UIDeviceFamily']
```

有効な XCtest UI パッケージでは、次のような出力が生成されます。

```
[1, 2]
```

詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_PACKAGE_NAME_VALUE_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist ファイルにパッケージ名の値が見つかりませんでした。テストパッケージを解凍し、次に .app ディレクトリ内の Info.plist ファイルを開き、キー "CFBundleIdentifier" が指定されていることを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

```
$ unzip swift-sample-UI.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

```
$ tree .
```

この例の *swift-sampleUITests-Runner.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあるはずです。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

3. パッケージ名の値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

Python の場合、次のコマンドを実行して biplist モジュールをインストールできます。

```
$ pip install biplist
```

4. 次に、Python を開き、次のコマンドを入力します。

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['CFBundleIdentifier']
```

有効な XCtest UI パッケージでは、次のような出力が生成されます。

```
com.apple.test.swift-sampleUITests-Runner
```

詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_EXECUTABLE_VALUE_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> Info.plist ファイルに実行可能な値が見つかりませんでした。テストパッケージを解凍し、次に .app ディレクトリ内の Info.plist ファイルを開き、キー "CFBundleExecutable" が指定されていることを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

```
$ unzip swift-sample-UI.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

```
$ tree .
```

この例の *swift-sampleUITests-Runner.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあるはずです。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

3. 実行可能な値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

   Python の場合、次のコマンドを実行して biplist モジュールをインストールできます。

   ```
   $ pip install biplist
   ```

4. 次に、Python を開き、次のコマンドを入力します。

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
   print info_plist['CFBundleExecutable']
   ```

   有効な XCtest UI パッケージでは、次のような出力が生成されます。

   ```
   XCTRunner
   ```

   詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_TEST_PACKAGE_NAME_VALUE_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> .xctest ディレクトリ内の Info.plist ファイルにパッケージ名の値が見つかりませんでした。テストパッケージを解凍し、次に .xctest ディレクトリ内の Info.plist ファイルを開き、キー "CFBundleIdentifier" が指定されていることを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

```
$ unzip swift-sample-UI.ipa
```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

```
$ tree .
```

この例の *swift-sampleUITests-Runner.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあるはずです。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                    |-- Info.plist
                    |-- Plugins (directory)
                    |       `swift-sampleUITests.xctest (directory)
                    |                       |-- Info.plist
                    |                       `-- (any other files)
                    `-- (any other files)
```

3. パッケージ名の値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

Python の場合、次のコマンドを実行して biplist モジュールをインストールできます。

```
$ pip install biplist
```

4. 次に、Python を開き、次のコマンドを入力します。

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Plugins/
swift-sampleUITests.xctest/Info.plist')
print info_plist['CFBundleIdentifier']
```

有効な XCtest UI パッケージでは、次のような出力が生成されます。

```
com.amazon.swift-sampleUITests
```

詳細については、「XCTest UI」を参照してください。

XCTEST_UI_TEST_PACKAGE_TEST_EXECUTABLE_VALUE_MISSING

次のメッセージが表示された場合は、次の手順に従って問題を解決してください。

> **Warning**
>
> .xctest ディレクトリ内の Info.plist ファイルに実行可能な値が見つかりませんでした。テストパッケージを解凍し、次に .xctest ディレクトリ内の Info.plist ファイルを開き、キー "CFBundleExecutable" が指定されていることを確認して、もう一度試してください。

次の例では、パッケージ名は swift-sample-UI.ipa です。

1. テストパッケージを作業ディレクトリにコピーし、次のコマンドを実行します。

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. 正常にパッケージを解凍したら、次のコマンドを実行して作業ディレクトリのツリー構造を見つけることができます。

   ```
   $ tree .
   ```

   この例の *swift-sampleUITests-Runner.app* のような *.app* ディレクトリ内に *Info.plist* ファイルがあるはずです。

   ```
   .
   `-- Payload (directory)
   ```

```
        `-- swift-sampleUITests-Runner.app (directory)
                    |-- Info.plist
                    |-- Plugins (directory)
                    |       `swift-sampleUITests.xctest (directory)
                    |                       |-- Info.plist
                    |                       `-- (any other files)
                    `-- (any other files)
```

3. 実行可能な値を見つけるため、Xcode または Python を使用して Info.plist を開くことができます。

   Python の場合、次のコマンドを実行して biplist モジュールをインストールできます。

   ```
   $ pip install biplist
   ```

4. 次に、Python を開き、次のコマンドを入力します。

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Plugins/
   swift-sampleUITests.xctest/Info.plist')
   print info_plist['CFBundleExecutable']
   ```

   有効な XCtest UI パッケージでは、次のような出力が生成されます。

   ```
   swift-sampleUITests
   ```

   詳細については、「XCTest UI」を参照してください。

AWS Device Farm のセキュリティ

AWS ではクラウド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、およびクラウド内でのセキュリティと説明しています:

- クラウドのセキュリティ - AWS は、AWS クラウドで AWS 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負います。また AWS は、安全に使用できるサービスを提供します。AWS コンプライアンスプログラムの一環として、サードパーティー監査者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ます。AWS Device Farm に適用され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で対象となる AWS 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 サービスに応じて異なります。またお客様は、データの機密性、企業要件、適用される法令と規制などの他の要因も責務となります。

このドキュメントは、Device Farm 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 Device Farm を構成する方法を示します。また、Device Farm リソースのモニタリングや保護に役立つ他の 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 Device Farm のアイデンティティとアクセス管理

対象者

AWS Identity and Access Management (IAM) の使用方法は、Device Farm で行う作業によって異なります。

サービスユーザー – ジョブを実行するために Device Farm サービスを使用する場合は、管理者から必要な認証情報と権限が与えられます。作業を実行するためにさらに多くの Device Farm の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しておくと、管理者に適切な権限をリクエストするうえで役立ちます。Device Farm の機能にアクセスできない場合は、「AWS Device Farm のアイデンティティとアクセス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 Device Farm リソースを担当している場合は、通常、Device Farm へのフルアクセス権があります。サービスユーザーがどの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点検して、IAM の基本概念を理解してください。貴社が Device Farm で IAM を利用する方法の詳細については、「AWS Device Farm と IAM の連携方法」を参照してください。

IAM 管理者 - 管理者は、Device Farm へのアクセスを管理するポリシーの記述方法についての詳細を学ぶとよいでしょう。IAM で使用できる Device Farm のアイデンティティベースポリシーの例を表示するには、「AWS Device Farm 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。

アイデンティティによる認証

認証とは、ID 認証情報 AWS を使用して にサインインする方法です。として、IAM ユーザーとして AWS アカウントのルートユーザー、または IAM ロールを引き受けて認証（にサインイン AWS) される必要があります。

ID ソースを介して提供された認証情報を使用して、フェデレーティッド ID AWS として にサインインできます。 AWS IAM Identity Center（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 の認証情報は、フェデレーティッド ID の例です。フェデレーティッド ID 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 AWS を使用して にアクセスすると、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。へのサインインの詳細については AWS、「 ユーザーガイド」の「 にサインインする方法 AWS アカウントAWS サインイン 」を参照してください。

AWS プログラムで にアクセスする場合、 は Software Development Kit (SDK) とコマンドラインインターフェイス (CLI) AWS を提供し、認証情報を使用して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。推奨される方法を使用してリクエストを自分で署名する方法の詳細については、IAM ユーザーガイドの API AWS 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、追加セキュリティ情報の提供をリクエストされる場合もあります。例えば、 AWS では、多要素認証 (MFA) を使用してアカウントのセキュリティを向上させることをお勧めします。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 ルートユーザー

を作成するときは AWS アカウント、アカウント内のすべての およびリソースへの AWS のサービス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 ID は AWS アカウント 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でアクセスできます。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、単一のユーザーまたはアプリケーションに対して特定のアクセス許可 AWS アカウント を持つ 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許可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、特定のアクセス許可 AWS アカウント を持つ 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 AWS Management Console ことで、 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、 または AWS API AWS CLI 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。

IAM ロールと一時的な認証情報は、次の状況で役立ちます:

- フェデレーションユーザーアクセス – フェデレーティッド ID に許可を割り当てるには、ロールを作成してそのロールの許可を定義します。フェデレーティッド ID が認証されると、その ID はロールに関連付けられ、ロールで定義されている許可が付与されます。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「Creating a role for a third-party Identity Provider」(サードパーティーアイデンティティプロバイダー向けロールの作成) を参照してください。IAM Identity Center を使用する場合は、許可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。アクセス許可セットの詳細については、「AWS IAM Identity Center ユーザーガイド」の「アクセス許可セット」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 では AWS のサービス、 (ロールをプロキシとして使用する代わりに) ポリシーをリソースに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 でのクロスアカウントのリソースへのアクセス」を参照してください。

- クロスサービスアクセス — 一部の は、他の の機能 AWS のサービス を使用します AWS のサービス。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許可、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 でアクションを実行する場合 AWS、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、 を呼び出すプリンシパルのアクセス許可を AWS のサービス、ダウンストリームサービス AWS のサービス へのリクエストのリクエストと組み合わせて使用します。FAS リクエストは、サービスが他の AWS のサービス またはリソースとのやり取りを完了する必要があるリクエストを受け取った場合にのみ行われます。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。
  - サービスにリンクされたロール – サービスにリンクされたロールは、 にリンクされたサービスロールの一種です AWS のサービス。サービスは、ユーザーに代わってアクションを実行するロールを引き受けることができます。サービスにリンクされたロールは に表示され AWS アカウント 、サービスによって所有されます。IAM 管理者は、サービスにリンクされたロールのアクセス許可を表示できますが、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 – IAM ロールを使用して、EC2 インスタンスで実行され、 AWS CLI または AWS API リクエストを行うアプリケーションの一時的な認証情報を管理できます。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、インスタンスにアタッチ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「(ユーザーの代わりに) IAM ロールを作成すべきとき」を参照してください。

AWS Device Farm と IAM の連携方法

Device Farm へのアクセス権を管理するために IAM を使用する前に、Device Farm でどの IAM 機能が使用できるかを理解しておく必要があります。Device Farm およびその他の 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AWS 「IAM と連携する のサービス」を参照してください。

トピック

- Device Farm のアイデンティティベースポリシー
- Device Farm のリソースベースポリシー
- アクセスコントロールリスト
- Device Farm タグに基づく認証
- Device Farm IAM ロール

Device Farm のアイデンティティベースポリシー

IAM のアイデンティティベースポリシーでは、許可または拒否するアクションとリソース、およびアクションが許可または拒否される条件を指定できます。Device Farm 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ています。JSON 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 ポリシー要素のリファレンス」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んなリソースにどんな条件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、関連付けられた 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。

関連付けられたオペレーションを実行する権限を付与するポリシーでのアクションを含みます。

Device Farm のポリシーアクションは、アクションの前に以下のプレフィックス「`devicefarm:`」を使用します: 例えば、Device Farm デスクトップブラウザテスト `CreateTestGridUrl` API

オペレーションで Selenium セッションを開始する権限を誰かに付与するには、そのポリシーに `devicefarm:CreateTestGridUrl` アクションを含めます。ポリシーステートメントには、`Action` または `NotAction` 要素を含める必要があります。Device Farm は、このサービスで実行できるタスクを記述する独自のアクションセットを定義します。

単一ステートメントに複数アクションを指定するには、次のようにカンマで区切ります:

```
"Action": [
      "devicefarm:action1",
      "devicefarm:action2"
```

ワイルドカード (*) を使用して複数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": "devicefarm:List*"
```

Device Farm アクションのリストを確認するには、「IAM サービス認可リファレンス」の「AWS Device Farmにより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どのような条件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、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、リソースレベルの権限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": "*"
```

Amazon EC2 インスタンスのリソースには次のような ARN があります:

```
arn:${Partition}:ec2:${Region}:${Account}:instance/${InstanceId}
```

ARN の形式の詳細については、「Amazon リソースネーム (ARNs AWS 「サービス名前空間」を参照してください。

例えば、ステートメントで i-1234567890abcdef0 インスタンスを指定するには、次の ARN を使用します:

```
"Resource": "arn:aws:ec2:us-east-1:123456789012:instance/i-1234567890abcdef0"
```

アカウントに属するすべてのインスタンスを指定するには、ワイルドカード (*) を使用します:

```
"Resource": "arn:aws:ec2:us-east-1:123456789012:instance/*"
```

リソースの作成など、一部の Device Farm アクションは、リソースで実行できません。このような場合は、ワイルドカード *を使用する必要があります。

```
"Resource": "*"
```

Amazon EC2 API アクションの多くが複数のリソースと関連します。例えば、AttachVolume では Amazon EBS ボリュームをインスタンスにアタッチするため、IAM ユーザーはボリュームおよびインスタンスを使用する権限が必要です。複数リソースを単一ステートメントで指定するには、ARN をカンマで区切ります。

```
"Resource": [
      "resource1",
      "resource2"
```

Device Farm リソースのタイプとその ARN のリストを確認するには、「IAM サービス認可リファレンス」の「AWS Device Farmにより定義されるリソース」を参照してください。各リソースの ARN を指定できるアクションについては、「IAM サービス認可リファレンス」の「AWS Device Farmにより定義されるアクション」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んなリソースにどんな条件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。1 つの条件キーに複数の値を指定すると、 は論理ORオペレーションを使用して条件 AWS を評価します。ステートメントの権限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権限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。

Device Farm は独自の条件キーセットを定義し、一部のグローバル条件キーの使用も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AWS 」の「グローバル条件コンテキストキー」を参照してください。

Shield の条件キーのリストを確認するには、「IAM サービス認証リファレンス」の「AWS Device Farmの条件キー」を参照してください。条件キーを使用できるアクションとリソースについては、「IAM サービス認証リファレンス」の「AWS Device Farmにより定義されるアクション」を参照してください。

例

Device Farm のアイデンティティベースポリシーの例を表示するには、「AWS Device Farm 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。

Device Farm のリソースベースポリシー

Device Farm では、リソースベース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

Device Farm では、アクセスコントロールリスト (ACL) はサポートされていません。

Device Farm タグに基づく認証

タグを Device Farm リソースにアタッチしたり、Device Farm へのリクエストでタグを渡したりできます。タグに基づいてアクセスを管理するには、aws:ResourceTag/*key-*

name、aws:RequestTag/*key-name*、または aws:TagKeys の条件キーを使用して、ポリシーの「条件要素」でタグ情報を提供します。Device Farm リソースのタギングの詳細については、「Device Farm でのタギング」を参照してください。

リソースのタグに基づいてリソースへのアクセスを制限するためのアイデンティティベースポリシーの例を表示するには、「タグに基づく Device Farm デスクトップブラウザテストプロジェクトの表示」を参照してください。

Device Farm 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。

Device Farm での一時的な認証情報の使用

Device Farm は、一時的な認証情報の使用をサポートしています。

一時的な認証情報を使用して、フェデレーションでサインインし、IAM ロールまたはクロスアカウントロールの引き受けを行えます。一時的なセキュリティ認証情報を取得するには、 AssumeRoleやなどの AWS STS API オペレーションを呼び出しますGetFederationToken。

サービスリンクロール

サービスにリンクされたロールを使用すると、 AWS サービスは他の サービスのリソースにアクセスして、ユーザーに代わってアクションを実行できます。サービスリンクロールは、IAM アカウント内に表示され、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。

Device Farm は、Device Farm デスクトップブラウザテスト機能でサービスリンクロールを使用します。これらのロールの詳細については、開発者ガイドの「Device Farm デスクトップブラウザテストのサービスリンクロールの使用」を参照してください。

サービスロール

Device Farm はサービスロールをサポートしていません。

この機能により、お客様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、サービスがお客様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 管理者は、このロールの権限を変更できます。ただし、それにより、サービスの機能が損なわれる場合があります。

ポリシーを使用したアクセス権の管理

でアクセスを制御する AWS には、ポリシーを作成し、 AWS ID またはリソースにアタッチします。ポリシーは、アイデンティティまたはリソースに関連付けられているときにアクセス許可を定義する のオブジェクトです。 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) AWS がリクエストを行うときに、これらのポリシー AWS 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。ほとんどのポリシーは JSON ドキュメント AWS として 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 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んなリソースにどんな条件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、アクションの許可を定義します。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。そのポリシーを持つユーザーは、 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のポリシー

アイデンティティベースポリシーは、IAM ユーザーグループ、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 許可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、および条件をコントロールします。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のポリシーは、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ています。管理ポリシーは、 内の複数のユーザー、グループ、ロールにアタッチできるスタンドアロンポリシーです AWS アカウント。管理ポリシーには、 AWS 管理ポリシーとカスタマー管理ポリシーが含まれます。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との間の選択」を参照してください。

次の表で、Device Farm AWS 管理ポリシーの概要を説明します。

変更	説明	日付
AWSDeviceFarmFullAccess	すべての AWS Device Farm オペレーションへのフルアクセスを提供します。	2015 年 7 月 15 日
AWSServiceRoleForDeviceFarmTestGrid	代わりに、Device Farm で AWS リソースにアクセスできます。	2021 年 5 月 20 日

他のポリシータイプ

AWS は、一般的ではない追加のポリシータイプをサポートします。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。

- アクセス許可の境界 - アクセス許可の境界は、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) に付与できる権限の上限を設定する高度な機能です。エンティティにアクセス許可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アクセス許可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、アクセス許可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。アクセス許可の境界の詳細については、IAM ユーザーガイドのIAM エンティティのアクセス許可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 (SCPs) – SCPs は、 の組織または組織単位 (OU) に対する最大アクセス許可を指定する JSON ポリシーです AWS Organizations。 AWS Organizations は、 AWS アカウント ビジネスが所有する複数の をグループ化して一元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、各 を含むメンバーアカウントのエンティティのアクセス許可を制限します AWS アカウントのルートユーザー。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参照してください。

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与する場合にリクエストを許可するかどうか AWS を決定する方法については、IAM ユーザーガイドの「ポリシー評価ロジック」を参照してください。

AWS Device Farm アイデンティティベースポリシーの例

デフォルトでは、IAM ユーザーおよびロールには、Device Farm リソースを作成したり変更したりする権限はありません。また、、 AWS Management Console AWS CLI、または AWS API を使用してタスクを実行することはできません。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権限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、管理者はそれらの権限が必要な IAM ユーザーまたはグループにそのポリシーをアタッチする必要があります。

JSON ポリシードキュメントのこれらの例を使用して、IAM 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。

トピック

- ポリシーのベストプラクティス
- ユーザーが自分の権限を表示できるようにする
- 1 つの Device Farm デスクトップブラウザテストプロジェクトへのアクセス
- タグに基づく Device Farm デスクトップブラウザテストプロジェクトの表示

ポリシーのベストプラクティス

アイデンティティベースポリシーは、ユーザーのアカウントで誰かが Device Farm 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では、 AWS アカウントに費用が発生する場合があります。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 管理ポリシーを開始し、最小特権のアクセス許可に移行する – ユーザーとワークロードにアクセス許可を付与するには、多くの一般的なユースケースにアクセス許可を付与する AWS 管理ポリシーを使用します。これらは で使用できます AWS アカウント。ユースケースに固有の AWS カスタマー管理ポリシーを定義して、アクセス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「ジョブ機能のAWS マネージドポリシー」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場合は、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できます。条件を使用して、 などの特定の を介してサービスアクションが使用される場合に AWS のサービス、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 AWS CloudFormation。詳細については、「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、安全で機能的な権限を確保する - 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 アクセスアナライザーは 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の作成をサポートします。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) を要求する – で IAM ユーザーまたはルートユーザーを必要とするシナリオがある場合は AWS アカウント、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 でのセキュリティベストプラクティス」を参照してください。

ユーザーが自分の権限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アタッチ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の作成方法を示します。このポリシーには、コンソールで、または AWS CLI または AWS API を使用してプログラムでこのアクションを実行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": "*"
        }
    ]
}
```

1 つの Device Farm デスクトップブラウザテストプロジェクトへのアクセス

この例では、 AWS アカウントの IAM ユーザーに Device Farm のデスクトップブラウザテストプロジェクトの 1 つである へのアクセス権を付与しますarn:aws:devicefarm:us-west-2:111122223333:testgrid-project:123e4567-e89b-12d3-a456-426655441111。アカウントでそのプロジェクトに関連するアイテムを表示できるようにする必要があります。

devicefarm:GetTestGridProject エンドポイントに加えて、アカウントには devicefarm:ListTestGridSessions、devicefarm:GetTestGridSession、devicefarm:ListTe および devicefarm:ListTestGridSessionArtifacts エンドポイントが必要です。

```
{
```

```
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"GetTestGridProject",
         "Effect":"Allow",
         "Action":[
            "devicefarm:GetTestGridProject"
         ],
         "Resource":"arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655441111"
      },
      {
         "Sid":"ViewProjectInfo",
         "Effect":"Allow",
         "Action":[
            "devicefarm:ListTestGridSessions",
            "devicefarm:ListTestGridSessionActions",
            "devicefarm:ListTestGridSessionArtifacts"
         ],
         "Resource":"arn:aws:devicefarm:us-west-2:111122223333:testgrid-*:123e4567-
e89b-12d3-a456-426655441111/*"
      }
   ]
}
```

CI システムを使用している場合は、各 CI 実行者に一意のアクセス認証情報を付与する必要があります。例えば、CI システムでは、devicefarm:ScheduleRun または devicefarm:CreateUpload よりも多くの権限が必要になることはほとんどありません。以下の IAM ポリシーでは、アップロードを作成してそのアップロードによりテスト実行をスケジュールすることで、新しい Device Farm ネイティブアプリケーションテストを開始することを、CI 実行者に許可する最小限のポリシーを概説します:

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "$id":"scheduleTestRuns",
         "effect":"Allow",
         "Action": [ "devicefarm:CreateUpload","devicefarm:ScheduleRun" ],
         "Resource": [
            "arn:aws:devicefarm:us-west-2:111122223333:project:123e4567-e89b-12d3-
a456-426655440000",
```

```
            "arn:aws:devicefarm:us-west-2:111122223333:*:123e4567-e89b-12d3-
a456-426655440000/*",
            ]
         }
    ]
}
```

タグに基づく Device Farm デスクトップブラウザテストプロジェクトの表示

アイデンティティベースポリシーの条件を使用して、タグに基づいて Device Farm リソースへのアクセスを管理できます。この例では、プロジェクトとセッションの表示を許可するポリシーを作成する方法を示しています。リクエスト先のリソースの Owner タグがリクエスト元のアカウントのユーザー名と一致する場合、権限が付与されます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ListTestGridProjectSessions",
            "Effect": "Allow",
            "Action": [
              "devicefarm:ListTestGridSession*",
              "devicefarm:GetTestGridSession",
              "devicefarm:ListTestGridProjects"
              ],
            "Resource": [
               "arn:aws:devicefarm:us-west-2:testgrid-project:*/*"
               "arn:aws:devicefarm:us-west-2:testgrid-session:*/*"
               ],
            "Condition": {
               "StringEquals": {"aws:TagKey/Owner":"${aws:username}"}
            }
        }
    ]
}
```

このポリシーはアカウントの IAM ユーザーにアタッチできます。richard-roe という名前のユーザーが Device Farm プロジェクトまたはセッションを表示しようとする場合、プロジェクトに Owner=richard-roe または owner=richard-roe というタグが付いている必要があります。それ以外の場合、ユーザーはアクセスを拒否されます。条件キー名では大文字と小文字は区別されない

ため、条件タグキー Owner は Owner と owner に一致します。詳細については、「IAM ユーザーガイド」の「IAM JSON ポリシー要素: 条件」を参照してください。

AWS Device Farm のアイデンティティとアクセスのトラブルシューティング

次の情報は、Device Farm と IAM による作業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。

Device Farm でアクションを実行する権限がない

で、アクションを実行する権限がない AWS Management Console というエラーが表示された場合は、管理者に連絡し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 という IAM ユーザーがコンソールを使用して、実行の詳細を表示しようとしているが、devicefarm:GetRun 権限がない場合に発生します。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 devicefarm:GetRun on resource: arn:aws:devicefarm:us-west-2:123456789101:run:123e4567-
e89b-12d3-a456-426655440000/123e4567-e89b-12d3-a456-426655441111
```

この場合、Mateo は管理者に依頼し、devicefarm:GetRun アクションを使用して arn:aws:devicefarm:us-west-2:123456789101:run:123e4567-e89b-12d3-a456-426655440000/123e4567-e89b-12d3-a456-426655441111 リソースに対する devicefarm:GetRun にアクセスできるようにポリシーを更新してもらいます。

iam を実行する権限がありません。PassRole

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 Device Farm 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またはサービスにリンクされたロールを作成する代わりに、そのサービスに既存のロールを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権限が必要です。

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 Device Farm でアクションを実行しようと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権限が必要です。メアリーには、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、Mary 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 管理者にお問い合わせください。サインイン認証情報を提供した担当者が管理者です。

アクセスキーを表示したい

IAM ユーザーアクセスキーを作成した後は、いつでもアクセスキー ID を表示できます。ただし、シークレットアクセスキーを再表示することはできません。シークレットアクセスキーを紛失した場合は、新しいアクセスキーペアを作成する必要があります。

アクセスキーは、アクセスキー ID (例: AKIAIOSFODNN7EXAMPLE) とシークレットアクセスキー (例: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY) の 2 つで構成されています。ユーザー名とパスワードと同様に、リクエストを認証するために、アクセスキー ID とシークレットアクセスキーの両方を使用する必要があります。ユーザー名とパスワードと同様に、アクセスキーは安全に管理してください。

> **Important**
>
> 正規のユーザー ID を確認するためであっても、アクセスキーを第三者に提供しないでください。これにより、 への永続的なアクセス権を誰かに付与できます AWS アカウント。

アクセスキーペアを作成する場合、アクセスキー ID とシークレットアクセスキーを安全な場所に保存するように求めるプロンプトが表示されます。このシークレットアクセスキーは、作成時にのみ使用できます。シークレットアクセスキーを紛失した場合、IAM ユーザーに新規アクセスキーを追加する必要があります。アクセスキーは最大 2 つまで持つことができます。既に 2 つある場合は、新規キーペアを作成する前に、いずれかを削除する必要があります。手順を表示するには、IAM ユーザーガイドの「アクセスキーの管理」を参照してください。

管理者として Device Farm へのアクセスを他のユーザーに許可したい

Device Farm 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

またはアプリケーションは、そ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、Device Farm の適切な権限を付与するエンティティにポリシーを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 が委任する初期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 アカウント以外の人に Device Farm 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人が、リソースにアクセスするために使用できるロールを作成できます。ロールの引き受けを委託するユーザーを指定できます。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) をサポートするサービスの場合、それらのポリシーを使用して、リソースへのアクセスを付与できます。

詳細については、以下を参照してください:

- Device Farm がこれらの機能をサポートしているかどうかについては、「AWS Device Farm と IAM の連携方法」で参照できます。
- 所有 AWS アカウント している のリソースへのアクセスを提供する方法については、IAM ユーザーガイドの「所有 AWS アカウント している別の の IAM ユーザーへのアクセスを提供する」を参照してください。
- リソースへのアクセスをサードパーティー に提供する方法については AWS アカウント、IAM ユーザーガイドの「サードパーティー AWS アカウント が所有する へのアクセスを提供する」を参照してください。
- ID フェデレーションを介してアクセスを提供する方法については、IAM ユーザーガイドの外部で認証されたユーザー (ID フェデレーション) へのアクセスの許可を参照してください。
- 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使用方法の違いについては、「IAM ユーザーガイド」の「IAM でのクロスアカウントのリソースへのアクセス」を参照してください。

AWS Device Farm のコンプライアンス検証

サードパーティーの監査者は、複数の AWS Device Farm コンプライアンスプログラムの一環として AWS のセキュリティとコンプライアンスを評価します。これらのプログラムとしては SOC、PCI、FedRAMP、HIPAA などがあります。AWS Device Farm は AWS コンプライアンスプログラムの対象ではありません。

特定のコンプライアンスプログラムの対象となる AWS サービスのリストについては、「コンプライアンスプログラムによる AWS 対象範囲内のサービス」を参照してください。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 および を参照してください。

Device Farm 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以下のリソースを提供しています。

- セキュリティおよび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 でデプロイするための手順を説明します。
- AWS コンプライアンスリソース – このワークブックおよびガイドのコレクションは、お客様の業界と拠点に適用される場合があります。
- 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 は、リソース設定が、社内のプラクティス、業界のガイドラインそして規制にどの程度適合しているのかを評価します。
- AWS Security Hub – この AWS のサービスは、AWS 内でのユーザーのセキュリティ状態に関する包括的な見解を提供し、業界のセキュリティ標準、およびベストプラクティスに対するコンプライアンスを確認するために役立ちます。

AWS Device Farm でのデータ保護

AWS 責任共有モデルは、AWS Device Farm (Device Farm) 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 には、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責任があります。ユーザーには、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機密情報やセンシティブ情報は、タグや、[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 を使用して、Device Farm または他の AWS のサービス で作業する場合も同様です。タグや、名前に使用する自由記述のテキストフィールドに入力したデータは、課金や診断ログに使用される場合があります。外部サーバーへ URL を提供する場合は、そのサーバーへのリクエストを有効にするために認証情報を URL に含めないことを強くお勧めします。

転送中の暗号化

Device Farm エンドポイントは、特に明記されていない限り、署名された HTTPS (SSL/TLS) リクエストのみをサポートしています。アップロード URL を通じて Amazon S3 が取得元または配置先となるすべてのコンテンツは、SSL/TLS を使用して暗号化されます。HTTPS リクエストの AWS での署名方法の詳細については、「AWS 一般的リファレンス」の「AWS API リクエストの署名」を参照してください。

テスト対象のアプリケーションが行う通信と、デバイスでのテストの実行中にインストールされるアプリケーションを暗号化して保護するのは、お客様の責任です。

保管中の暗号化

Device Farm のデスクトップブラウザーテスト機能は、テスト中に生成されたアーティファクトの保存時の暗号化をサポートします。

Device Farm の物理的モバイルデバイスのテストデータは保管時に暗号化されません。

データ保持

Device Farm のデータは一定期間保持されます。保持期間が終了すると、データは Device Farm バッキングストレージから削除されますが、メタデータ (ARN、アップロード日、ファイル名など) は今後の使用のために保持されます。以下の表では、さまざまなコンテンツタイプの保持期間を示しています。

コンテンツタイプ	保持期間 (日数)
アップロードされたアプリケーション	30
アップロードされたテストパッケージ	30
ログ	400
ビデオ録画や他のアーティファクト	400

長期間保持するコンテンツをアーカイブするのは、お客様の責任です。

データ管理

Device Farm のデータは、使用する機能に応じて異なる方法で管理されます。このセクションでは、Device Farm の使用中および使用後のデータの管理方法について説明します。

デスクトップブラウザテスト

Selenium セッション中に使用されるインスタンスは保存されません。ブラウザ操作の結果として生成されたすべてのデータは、セッションが終了すると破棄されます。

この機能は現在、テスト中に生成されたアーティファクトの保存時の暗号化をサポートしています。

物理デバイスのテスト

次のセクションでは、Device Farm を使用した後にデバイスをクリーンアップまたは破壊するために AWS が行うステップについて説明します。

Device Farm の物理的モバイルデバイスのテストデータは保管時に暗号化されません。

パブリックデバイスフリート

テスト実行が完了すると、Device Farm はパブリックデバイスフリートの各デバイスで、一連のクリーンアップタスク (アプリケーションのアンインストールなど) を実行します。アプリケーションのアンインストールまたは他のいずれかのクリーンアップステップを確認できない場合、デバイスが再利用される前にファクトリのリセットが実行されます。

> **Note**
>
> 場合によっては (特に、アプリケーションのコンテキスト外で デバイスシステムを使用する場合)、セッション間でデータが保持されることがあります。また、Device Farm は各デバイスの使用中に行われるアクティビティのビデオとログをキャプチャするため、自動テストおよびリモートアクセスセッション中に、機密情報 (Google アカウント、Apple ID など)、個人情報、および他のセキュリティ上センシティブな詳細は入力しないことをお勧めします。

プライベートデバイス

プライベートデバイス契約の終了または解除後、デバイスは使用から除外され、AWS 破壊ポリシーに従って安全に破壊されます。詳細については、「AWS Device Farm でのプライベートデバイスによる作業」を参照してください。

キー管理

現在 Device Farm は、保管中または転送中のデータ暗号化用外部キー管理は提供していません。

インターネットトラフィックのプライバシー

Device Farm は、プライベートデバイスに対してのみ、Amazon VPC エンドポイントを使用して AWS のリソースに接続するように構成できます。アカウントに関連付けられた非パブリック AWS インフラストラクチャへのアクセス (パブリック IP アドレスのない Amazon EC2 インスタンスなど) には、Amazon VPC エンドポイントを使用する必要があります。VPC エンドポイントの構成に関係なく、Device Farm は Device Farm ネットワーク全体でトラフィックを他のユーザーから分離します。

AWS ネットワークの外部での接続が安全であることは保証されません。アプリケーションが行うインターネット接続を保護するのは、お客様の責任です。

AWS Device Farm での耐障害性

AWS のグローバルインフラストラクチャは AWS リージョンとアベイラビリティーゾーンを中心に構築されます。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 Connect されている複数の物理的に独立・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では、ゾーン間で中断することなく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することができます。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、耐障害性、および拡張性が優れています。

AWS リージョンとアベイラビリティーゾーンの詳細については、「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。

Device Farm は `us-west-2` リージョンでのみ利用できるため、バックアップおよび復旧プロセスを実装することを強くお勧めします。Device Farm は、アップロードされたコンテンツの唯一のソースであってはなりません。

Device Farm では、パブリックデバイスの可用性が保たれるとは限りません。これらのデバイスは、障害発生率や隔離ステータスなどのさまざまな要因に応じて、パブリックデバイスプールに対して追加または削除されます。パブリックデバイスプール内のいずれかのデバイスの可用性に依存することはお勧めしません。

AWS Device Farm 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 Device Farm は AWS グローバルネットワークセキュリティで保護されています。AWS セキュリティサービスと AWS がインフラストラクチャを保護する方法については、「AWS クラウドセキュリティ」を参照してください。インフラストラクチャセキュリティのベストプラクティスにより AWS 環境を設計するには、「セキュリティの柱 - AWS Well-Architected フレームワーク」の「インフラストラクチャ保護」を参照してください。

AWS 公開済み API コールを使用して、ネットワーク経由で Device Farm にアクセスします。クライアントは以下をサポートする必要があります:

- Transport Layer Security (TLS)。TLS 1.2 は必須で TLS 1.3 がお勧めです。
- DHE (楕円ディフィー・ヘルマン鍵共有) や ECDHE (楕円曲線ディフィー・ヘルマン鍵共有) などの完全前方秘匿性 (PFS) による暗号スイート。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。

また、リクエストには、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。

物理デバイステストのインフラストラクチャセキュリティ

物理デバイスのテスト中、デバイスは物理的に分離されています。ネットワークの分離により、ワイヤレスネットワークを介したデバイス間通信が防止されます。

パブリックデバイスは共有され、Device Farm はベストエフォートベースでデバイスを長期間にわたって安全に保ちます。デバイスに対する完全な管理者権限を取得する試み (ルート化または脱獄と呼ばれる行為) などの特定アクションが検出されると、パブリックデバイスが隔離されます。それらのデバイスは自動的にパブリックプールから削除され、手動点検の対象になります。

プライベートデバイスへのアクセスは明示的に承認された AWS アカウントでのみ可能です。Device Farm は、これらのデバイスを他のデバイスから物理的に隔離し、別のネットワークで保持します。

プライベートで管理されているデバイスでは、Amazon VPC エンドポイントを使用して AWS アカウントとの接続を保護するようにテストを構成できます。

デスクトップブラウザテストのインフラストラクチャセキュリティ

デスクトップブラウザテスト機能を使用すると、すべてのテストセッションが互いに分離されます。Selenium インスタンスは、AWS の外部にある仲介サードパーティーなしでは相互通信できません。

Selenium WebDriver コントローラーへのすべてのトラフィックは、`createTestGridUrl` で生成された HTTPS エンドポイントを通過する必要があります。

現時点では、デスクトップブラウザテスト機能は Amazon VPC エンドポイント構成をサポートしていません。各 Device Farm テストインスタンスがテスト対象のリソースに安全にアクセスできることを確認するのは、お客様の責任です。

Device Farm での構成の脆弱性の分析と管理

Device Farm を使用すると、OS ベンダー、ハードウェアベンダー、電話キャリアなど、ベンダーによってアクティブに保守またはパッチされていないソフトウェアを実行できます。Device Farm は、ベストエフォート型として、最新のソフトウェアを管理しようとしますが、その設計上、脆弱な可能性があるソフトウェアの使用を認めることがあり、物理デバイスの特定のソフトウェアのバージョンが最新であるとは保証しません。

例えば、Android 4.4.2 で動作しているデバイスでテストを実行しても、Device Farm は、StageFright と呼ばれる Android の脆弱性に対してデバイスがパッチされていることを保証しません。デバイスにセキュリティ更新を提供するのは、デバイスのベンダー (場合によってはキャリア) の責任です。また、この脆弱性を使用する悪意のあるアプリケーションが自動検疫で検出されることは保証されません。

プライベートデバイスは AWS との契約に従って管理されます。

Device Farm は、お客様のアプリケーションがルーティングまたはジェイルブレイキングなどのアクションを実行しないよう誠意ある最善の努力を行います。Device Farm は、パブリックプールから隔離されたデバイスを、手動で確認されるまで削除します。

お客様は、Python ホイールや Ruby Gem など、テストで使用するソフトウェアのライブラリやバージョンを最新に保つ責任があります。Device Farm では、テストライブラリを更新することをお勧めします。

これらのリソースは、テストの依存関係を最新に保つのに役立ちます:

- Ruby Gem を保護する方法については、RubyGems ウェブサイトの「セキュリティ慣行」を参照してください。
- 既知の脆弱性の依存関係グラフをスキャンするために Pipenv によって使用され、Python Packaging Authority によって承認される安全パッケージについては、GitHub の「セキュリティ脆弱性の検出」を参照してください。
- Open Web Application Security Project (OWASP) Maven 依存関係チェッカーについては、OWASP ウェブサイトの「OWASP DependencyCheck」を参照してください。

自動化システムによって既知のセキュリティ問題が検出されなくても、セキュリティ問題がないことを意味するわけではありません。サードパーティーのライブラリまたはツールを使用するときは、常にデューデリジェンスを使用し、可能または合理的な場合は、暗号化署名を点検してください。

Device Farm でのインシデント応答

Device Farm は、セキュリティの問題を示す可能性のある動作に関してデバイスを継続的にモニタリングします。お客様データ (テスト結果や、パブリックデバイスに書き込まれたファイルなど) に別のお客様がアクセスできるケースを AWS が認識した場合、AWS サービス全体で使用される標準のインシデントアラートおよびレポートポリシーに従って、AWS は影響を受けるお客様に連絡します。

Device Farm でのロギングとモニタリング

このサービスは、AWS アカウント への AWS の呼び出しを記録し、ログファイルを Amazon S3 バケットに配信するサービスである AWS CloudTrail をサポートします。CloudTrail が収集した情報を使えば、AWS のサービス に正常に行われたリクエストの内容、そのリクエストの送信者、そのリクエストの送信時刻などを判断できます。CloudTrail の詳細 (有効にする方法、ログファイルを検索する方法を含む) については、「AWS CloudTrail ユーザーガイド」を参照してください。

Device Farm による CloudTrail の使用についての詳細は、「AWS CloudTrail による AWS Device Farm API コールのロギング」を参照してください。

Device Farm のセキュリティベストプラクティス

Device Farm には、独自のセキュリティポリシーを策定および実装する際に考慮すべきさまざまなセキュリティ機能が備わっています。以下のベストプラクティスは一般的なガイドラインであり、完全なセキュリティソリューションに相当するものではありません。これらのベストプラクティスはお客様の環境に必ずしも適切または十分でない可能性があるので、処方箋ではなく、あくまで有用な考慮事項とお考えください。

- 使用する継続的統合 (CI) システムに、IAM で可能な最小限の権限を付与します。各 CI システムテストには一時的な認証情報を使用することを検討してください。これにより、CI システムが不正アクセスされても偽のリクエストを行えなくなります。一時的な認証情報についての詳細は、「IAM ユーザーガイド」を参照してください。
- カスタムテスト環境で adb コマンドを使用して、アプリケーションによって作成されたコンテンツをクリーンアップします。カスタムテスト環境の詳細については、「カスタムテスト環境による作業」を参照してください。

AWS Device Farm での制限

次のリストは、AWS Device Farm の現在の制限を示します。

- アップロードできるアプリケーションの最大ファイルサイズは 4 GB です。
- テスト実行に含めることができるデバイスの数に制限はありません。ただし、Device Farm がテスト実行中に同時にテストするデバイスの最大数は 5 です。(この数値は、リクエストに応じて増やすことができます。)
- スケジュールできる実行の数に制限はありません。
- リモートアクセスセッションの期間には、150 分の制限があります。
- 自動化されたテスト実行の時間には、150 分の制限があります。
- アカウントでキューに入っている保留中ジョブを含め、取り扱いジョブの最大数は250です。これはソフトリミットです。
- テストランに含めることができるデバイスの数に制限はありません。特定の時点においてテストを並列で実行できるデバイスまたはジョブの数は、アカウントレベルの同時実行数と同じです。AWS Device Farm での計測時の使用における、デフォルトのアカウントレベルの同時実行数は 5 です。ユースケースに応じて、この数を特定のしきい値まで増やすようリクエストできます。未計測時の使用におけるデフォルトのアカウントレベルの同時実行数は、そのプラットフォームでサブスクライブしているスロット数と同じです。

AWS Device Farm のツールとプラグイン

このセクションには、AWS Device Farm のツールとプラグインでの作業に関するリンクと情報が含まれています。Device Farm プラグインは、GitHub の AWS ラボにあります。

Android 開発者である場合は、「GitHub の Android 用 AWS Device Farm サンプルアプリケーション」も利用できます。このアプリケーションおよびテスト例は、独自の Device Farm テストスクリプトのリファレンスとして使用できます。

トピック

- AWS Device Farm の Jenkins CI プラグインとの統合
- AWS Device Farm Gradle プラグイン

AWS Device Farm の Jenkins CI プラグインとの統合

このプラグインによって、独自の Jenkins 継続的インテグレーション (CI) サーバーから AWS Device Farm 機能が提供されます。詳細については、「Jenkins (ソフトウェア)」を参照してください。

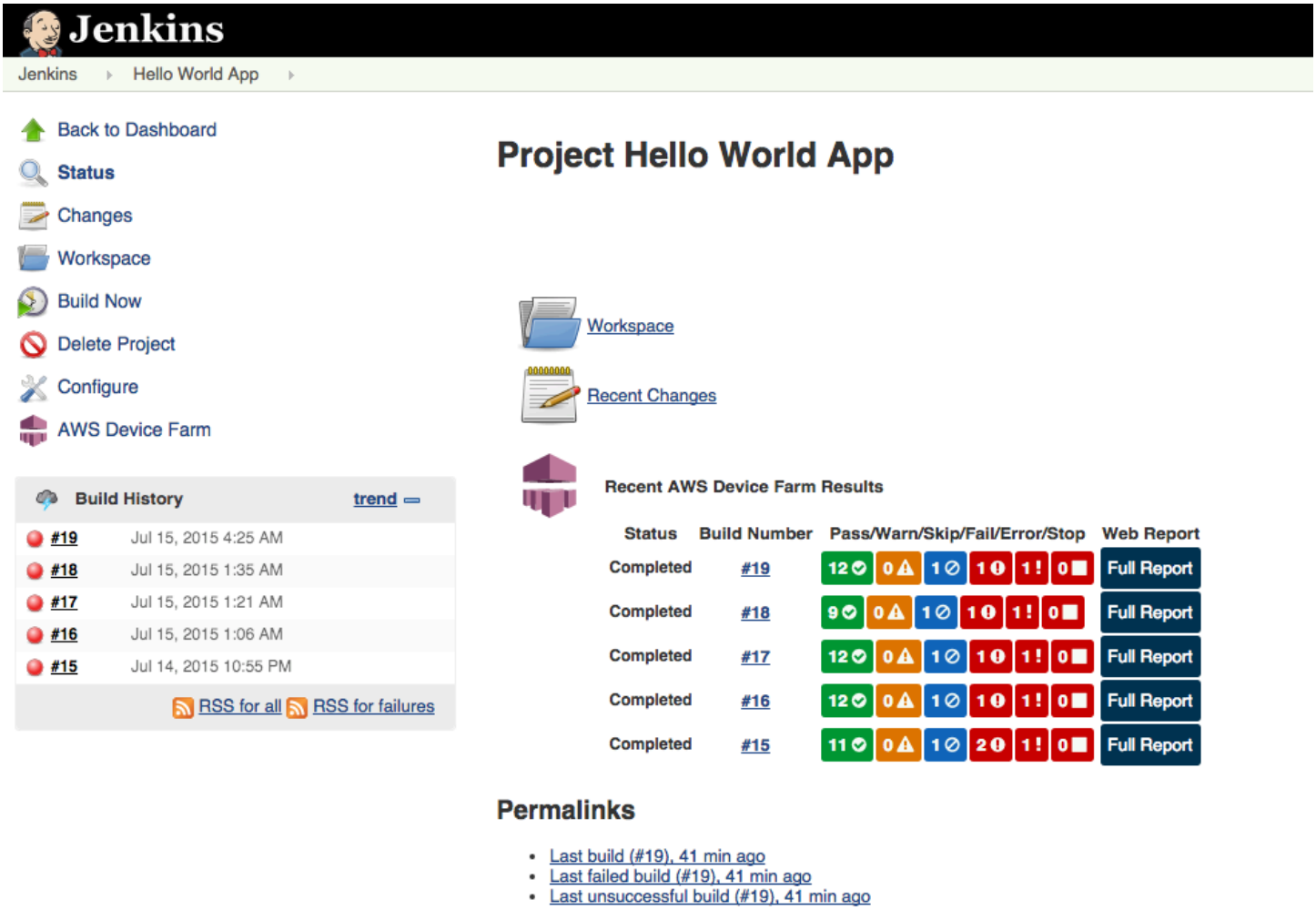
> **Note**
>
> Jenkins プラグインをダウンロードするには、GitHub にアクセスし、「ステップ 1: プラグインのインストール」の手順に従います。

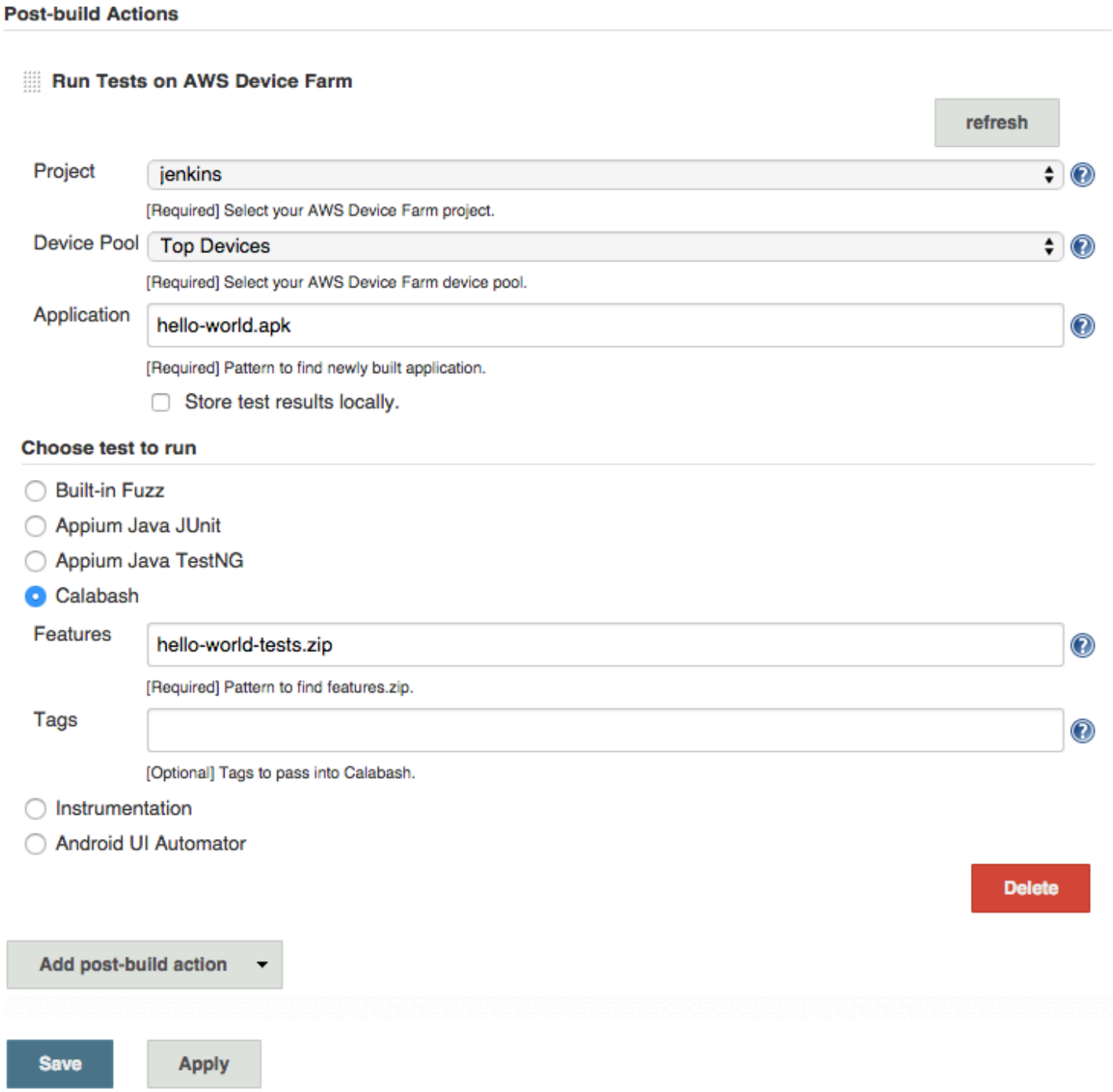
このセクションには、AWS Device Farm で Jenkins CI プラグインをセットアップして使用する一連の手順が含まれます。

トピック

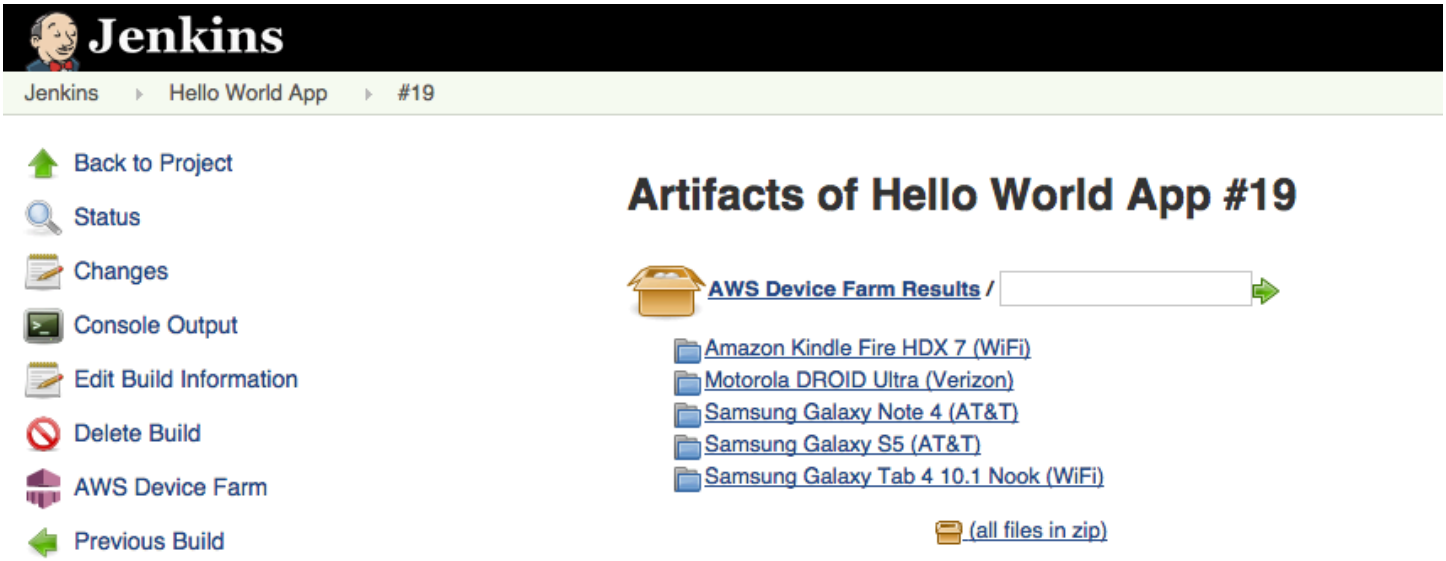
- ステップ 1: プラグインのインストール
- ステップ 2: Jenkins CI プラグインのための AWS Identity and Access Management ユーザーを作成する
- ステップ 3: 初回の設定手順
- ステップ 4: Jenkins ジョブでのプラグインの使用
- 依存関係

以下の画像は、Jenkins CI プラグインの機能を示しています。





また、このプラグインではすべてのテストアーティファクト (ログ、スクリーンショットなど) をローカルにプルダウンすることもできます。



ステップ 1: プラグインのインストール

AWS Device Farm の Jenkins 継続的インテグレーション (CI) プラグインをインストールするには 2 つのオプションがあります。Jenkins ウェブ UI の「使用できるプラグイン」ダイアログ内からプラグインを検索するか、hpi ファイルをダウンロードして Jenkins 内からインストールできます。

Jenkins UI 内からインストールする

1. [Jenkins を管理]、[プラグインを管理] を選択し、次に [使用可能] を選択して Jenkins UI 内でプラグインを見つけます。
2. aws-device-farm を検索します。
3. AWS Device Farm プラグインをインストールします。
4. プラグインが `Jenkins` ユーザーに所有されていることを確認します。
5. Jenkins を再起動します。

プラグインをダウンロードする

1. hpi ファイルを直接 http://updates.jenkins-ci.org/latest/aws-device-farm.hpi からダウンロードします。
2. プラグインが `Jenkins` ユーザーに所有されていることを確認します。
3. 次のいずれかのオプションを使用して、プラグインをインストールします:

- [Jenkins を管理]、[プラグインを管理]、[詳細]、[プラグインをアップロード] の順に選択してプラグインをアップロードします。
- hpi ファイルを Jenkins プラグインディレクトリ (通常は `/var/lib/jenkins/plugins`) に配置します。

4. Jenkins を再起動します。

ステップ 2: Jenkins CI プラグインのための AWS Identity and Access Management ユーザーを作成する

Device Farm へのアクセスには、AWS ルートアカウントは使用しないことをお勧めします。代わりに、AWS アカウントで新規 AWS Identity and Access Management (IAM) ユーザーを作成 (または既存の IAM ユーザーを使用) し、その IAM ユーザーで Device Farm にアクセスします。

新しい IAM ユーザーを作成するには、「IAM ユーザーの作成 (AWS Management Console)」を参照してください。各ユーザーのアクセスキーを生成していることを確認し、ユーザーのセキュリティ認証情報をダウンロードまたは保存します。認証情報は後で必要になります。

IAM ユーザーに Device Farm へアクセスする権限を付与します。

IAM ユーザーに Device Farm へアクセスする権限を付与するには、IAM で新規アクセスポリシーを作成し、そのアクセスポリシーを次のように IAM ユーザーに割り当てます。

> **Note**
>
> 次のステップを完了するために使用する AWS ルートアカウントまたは IAM ユーザーには、次の IAM ポリシーを作成し、その IAM ユーザーにアタッチする権限が付与されている必要があります。詳細については、「ポリシーによる作業」を参照してください。

IAM でアクセスポリシーを作成するには

1. IAM コンソール (https://console.aws.amazon.com/iam/) を開きます。
2. [ポリシー] を選択します。
3. [ポリシーを作成] を選択します。(開始する] ボタンが表示された場合は、そのボタンを選択してから、[ポリシーを作成] を選択します)。
4. [独自のポリシーを作成] の横で、[選択] を選択します。

5. [ポリシー名] に、ポリシーの名前 (**AWSDeviceFarmAccessPolicy** など) を入力します。
6. [説明] に、この IAM ユーザーを Jenkins プロジェクトに関連付けるための説明を入力します。
7. [ポリシードキュメント] に、次のステートメントを入力します:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DeviceFarmAll",
            "Effect": "Allow",
            "Action": [ "devicefarm:*" ],
            "Resource": [ "*" ]
        }
    ]
}
```

8. [ポリシーを作成] を選択します。

アクセスポリシーを IAM ユーザーに割り当てるには

1. IAM コンソール (https://console.aws.amazon.com/iam/) を開きます。
2. [ユーザー] を選択します。
3. アクセスポリシーを割り当てる IAM ユーザーを選択します。
4. [権限] エリアの [管理ポリシー] で、[ポリシーをアタッチ] を選択します。
5. 先ほど作成したポリシーを選択します (例: AWSDeviceFarmAccessPolicy)。
6. [ポリシーをアタッチ] を選択します。

ステップ 3: 初回の設定手順

Jenkins サーバーを初めて実行するときは、次のようにシステムを設定する必要があります。

> **Note**
>
> デバイススロットを使用している場合、デバイススロット機能はデフォルトで無効になっています。

1. Jenkins のウェブユーザーインターフェイスにログインします。

2. 画面の左側で、[Manage Jenkins] (Jenkins を管理) を選択します。
3. [Configure System] (システムを設定) を選択します。
4. [AWS Device Farm] ヘッダーまで下にスクロールします。
5. ステップ 2: IAM ユーザーを作成する からセキュリティ認証情報をコピーして、アクセスキー ID とシークレットアクセスキーをそれぞれのボックスに貼り付けます。
6. [Save (保存)] を選択します。

ステップ 4: Jenkins ジョブでのプラグインの使用

Jenkins プラグインのインストールが完了したら、次の手順に従って、Jenkins ジョブでプラグインを使用します。

1. Jenkins のウェブ UI にログインします。
2. 編集するジョブをクリックします。
3. 画面の左側にある [構成] を選択します。
4. [ビルド後アクション] ヘッダーまでスクロールダウンします。
5. [ビルド後アクションを追加] をクリックして [AWS Device Farm でテストを実行] を選択します。
6. 使用するプロジェクトを選択します。
7. 使用するデバイスプールを選択します。
8. テストアーティファクト (ログやスクリーンショットなど) をローカルでアーカイブするかどうかを選択します。
9. [アプリケーション] で、コンパイルされたアプリケーションへのパスを入力します。
10. 実行するテストを選択し、すべての必須フィールドに入力します。
11. [保存] を選択します。

依存関係

Jenkins CI プラグインには、AWS Mobile SDK 1.10.5 以降が必要です。SDK の詳細とインストールについては、「AWS Mobile SDK」を参照してください。

AWS Device Farm Gradle プラグイン

このプラグインによって、AWS Device Farm は Android Studio の Gradle ビルドシステムと統合されます。詳細については、「Gradle」を参照してください。

> **Note**
>
> Gradle Plugin をダウンロードするには、「GitHub」に移動し、Device Farm Gradle プラグインの構築の手順に従います。

Device Farm Gradle Plugin により、Android Studio 環境から Device Farm 機能が提供されます。Device Farm によってホストされる現行の Android 電話やタブレットでテストを開始できます。

このセクションには、Device Farm Gradle Plugin をセットアップして使用する一連の手順が含まれます。

トピック

ステップ 1: AWS Device Farm Gradle プラグインの構築

このプラグインによって、AWS Device Farm が Android Studio の Gradle ビルドシステムと統合されます。詳細については、「Gradle」を参照してください。

> **Note**
>
> プラグインの構築はオプションです。プラグインは、Maven Central を通じて発行されます。Gradle がプラグインを直接ダウンロードするよう許可する場合は、この手順をスキップして、ステップ 2: AWS Device Farm Gradle プラグインのセットアップに進みます。

プラグインを構築するには

1. 「GitHub」に移動して、リポジトリのクローンを作成します。
2. `gradle install`を使用してプラグインを構築します。

   プラグインはローカルの maven リポジトリにインストールされます。

次のステップ: ステップ 2: AWS Device Farm Gradle プラグインのセットアップ

ステップ 2: AWS Device Farm Gradle プラグインのセットアップ

リポジトリのクローン作成とプラグインのインストールをまだ行っていない場合は、以下の手順を参照して実行してください。Device Farm Gradle プラグインの構築

AWS Device Farm Gradle Pluginを構成するには

1. `build.gradle` の依存関係リストにプラグインアーティファクトを追加します。

```
    buildscript {

        repositories {
            mavenLocal()
            mavenCentral()
        }

        dependencies {
            classpath 'com.android.tools.build:gradle:1.3.0'
            classpath 'com.amazonaws:aws-devicefarm-gradle-plugin:1.0'
        }
    }
```

2. `build.gradle` ファイルのプラグインを構成します。以下のテスト固有の構成がガイドとして役立ちます。

```
apply plugin: 'devicefarm'

devicefarm {

    // Required. The project must already exist. You can create a project in the
 AWS Device Farm console.
    projectName "My Project" // required: Must already exist.
```

```
   // Optional. Defaults to "Top Devices"
   // devicePool "My Device Pool Name"

   // Optional. Default is 150 minutes
   // executionTimeoutMinutes 150

   // Optional. Set to "off" if you want to disable device video recording during
a run. Default is "on"
   // videoRecording "on"

   // Optional. Set to "off" if you want to disable device performance monitoring
during a run. Default is "on"
   // performanceMonitoring "on"

   // Optional. Add this if you have a subscription and want to use your unmetered
slots
   // useUnmeteredDevices()

   // Required. You must specify either accessKey and secretKey OR roleArn.
roleArn takes precedence.
   authentication {
       accessKey "AKIAIOSFODNN7EXAMPLE"
       secretKey "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"

       // OR

       roleArn "arn:aws:iam::111122223333:role/DeviceFarmRole"
   }

   // Optionally, you can
   // - enable or disable Wi-Fi, Bluetooth, GPS, NFC radios
   // - set the GPS coordinates
   // - specify files and applications that must be on the device when your test
runs
   devicestate {
       // Extra files to include on the device.
       // extraDataZipFile file("path/to/zip")

       // Other applications that must be installed in addition to yours.
       // auxiliaryApps files(file("path/to/app"), file("path/to/app2"))

       // By default, Wi-Fi, Bluetooth, GPS, and NFC are turned on.
       // wifi "off"
```

```
        // bluetooth "off"
        // gps "off"
        // nfc "off"

        // You can specify GPS location. By default, this location is 47.6204,
 -122.3491
        // latitude 44.97005
        // longitude -93.28872
    }

    // By default, the Instrumentation test is used.
    // If you want to use a different test type, configure it here.
    // You can set only one test type (for example, Calabash, Fuzz, and so on)

    // Fuzz
    // fuzz { }

    // Calabash
    // calabash { tests file("path-to-features.zip") }

}
```

3. 次のタスク (`gradle devicefarmUpload`) を使用して Device Farm テストを実行します。

   ビルド出力で Device Farm コンソールへのリンクが出力され、テストの実行をモニタリングできます。

次のステップ : IAM ユーザーの作成

ステップ 3: IAM ユーザーの作成

AWS Identity and Access Management (IAM) を使用して、AWS リソースによる作業のための権限とポリシーを管理できます。このトピックでは AWS Device Farm リソースにアクセスする権限を持つ IAM ユーザーの作成方法について説明します。

手順 1 と 2 を実行していない場合は、IAM ユーザーを作成する前に完了してください。

Device Farm へのアクセスには、AWS ルートアカウントを使用しないことをお勧めします。代わりに、AWS アカウントに新しい IAM ユーザーを作成 (または既存の IAM ユーザーを使用) し、その IAM ユーザーで Device Farm にアクセスします。

> **Note**
>
> 次のステップを完了するために使用する AWS ルートアカウントまたは IAM ユーザーは、次の IAM ポリシーを作成し、その IAM ユーザーにアタッチする権限を持つ必要があります。詳細については、「ポリシーを使った作業」を参照してください。

IAM で適切なアクセスポリシーを持つ、新しいユーザーを作成するには

1. IAM コンソール (https://console.aws.amazon.com/iam/) を開きます。
2. [ユーザー] を選びます。
3. [新規ユーザーを作成] を選びます。
4. 任意のユーザー名を入力します。

   たとえば、**GradleUser** と指定します。
5. [作成] を選びます。
6. [認証情報をダウンロード] を選び、後で簡単に取得できる場所に保存します。
7. [閉じる] を選びます。
8. リスト内でユーザー名を選びます。
9. [権限] で、右側にある下矢印をクリックして [インラインポリシー] ヘッダーを展開します。
10. [こちらをクリック] を選ぶと、「表示するインラインポリシーはありません」と表示されます。作成するには、ここをクリックしてください。
11. 「権限を設定」画面で [カスタムポリシー] を選びます。
12. [選択] を選びます。
13. ポリシーに名前を付けます (例: **AWSDeviceFarmGradlePolicy**)。
14. 次のポリシーを [ポリシードキュメント] に貼り付けます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DeviceFarmAll",
            "Effect": "Allow",
            "Action": [ "devicefarm:*" ],
            "Resource": [ "*" ]
        }
```

```
        ]
    }
```

15. [ポリシーを適用] を選びます。

次のステップ: 次は「テストタイプの構成」です。

詳細については、「IAM ユーザーの作成 (AWS Management Console)」または「設定」を参照してください。

ステップ 4: テストタイプの構成

デフォルトでは、AWS Device Farm Gradle プラグインはAndroid および AWS Device Farm のインストゥルメンテーションによる作業テストを実行します。独自のテストを実行、または追加のパラメーターを指定する場合は、テストタイプを構成できます。このトピックでは、利用可能な各テストタイプに関する情報と、使用のための構成に向けて Android Studio で行う必要がある内容について説明します。Device Farm で利用可能なテストタイプの詳細については、「AWS Device Farm のテストタイプによる作業」を参照してください。

まだ行っていない場合は、テストタイプを構成する前に手順 1～3 を完了します。

> **Note**
>
> デバイススロットを使用している場合、デバイススロット機能はデフォルトで無効になっています。

Appium

Device Farm は、Android 向け Appium Java JUnit および TestNG のサポートを提供します。

- Appium (Java (JUnit))
- Appium (Java (TestNG))

`useTestNG()` または `useJUnit()` を選択します。デフォルトとなる `JUnit` は、特別に指定する必要はありません。

```
    appium {
        tests file("path to zip file") // required
```

```
        useTestNG() // or useJUnit()
    }
```

Built-in: ファズ

Device Farm では、ランダムにユーザーインターフェイスイベントをデバイスに送信し、その結果をレポートする組み込みファズテストタイプが提供されます。

```
    fuzz {

       eventThrottle 50 // optional default
       eventCount 6000  // optional default
       randomizerSeed 1234 // optional default blank

     }
```

詳細については、「ビルトイン: ファズ (Android および iOS)」を参照してください。

インストルメンテーション

Device Farm では、Android 向けインストルメンテーション (JUnit、Espresso、Robotium、または任意の実装ベーステスト) のサポートが提供されます。詳細については、「Android および AWS Device Farm のインストゥルメンテーションによる作業」を参照してください。

Gradle でインストルメンテーションテストを実行する場合、Device Farm では、[androidTest] ディレクトリから生成された .apk ファイルをテストのソースとして使用します。

```
    instrumentation {

        filter "test filter per developer docs" // optional

    }
```

依存関係

ランタイム

- Device Farm Gradle Plugin には、AWS Mobile SDK 1.10.15 以降が必要です。SDK のインストールおよび詳細については、「AWS Mobile SDK」を参照してください。
- Android ツールビルダーテスト api 0.5.2

- Apache Commons Lang3 3.3.4

ユニットテストの場合

- Testng 6.8.8
- Jmockit 1.19
- Android Gradle 1.3.0 ツール

ドキュメント履歴

以下の表に、このガイドの前回のリリース以降に行われた重要な変更を示します。

変更	説明	変更日
AL2 のサポート	Device Farm は Android 用の AL2 テスト環境をサポートするようになりました。AL2 の詳細については、こちらを参照してください。	2023 年 11 月 6 日
標準テスト環境からカスタムテスト環境への移行	移行ガイドを更新し、2023 年 12 月に標準モードテストの廃止を文書化しました。	2023 年 9 月 3 日
VPC ENI のサポート	Device Farm では、プライベートデバイスが VPC-ENI 接続機能を使用して、お客様が AWS、オンプレミスソフトウェア、または別のクラウドプロバイダーでホストされているプライベートエンドポイントに安全に接続できるよう支援するようになりました。VPC-ENI の詳細については、こちらをご覧ください。	2023 年 5 月 15 日
Polaris UI の更新	Device Farm コンソール は、現在 Polaris フレームワークがサポートされます。	2021 年 7 月 28 日
Python 3 のサポート	Device Farm は、現在カスタムモードテストで Python 3 をサポートするようになりました。テストパッケージでの Python 3 の使用方法の詳細については、こちらを参照してください。 • Appium (Python) • Appium (Python)	2020 年 4 月 20 日
新しいセキュリティ情報と AWS リソースのタグ付けに関する情報。	AWS のサービスの保護をより容易かつ包括的にするために、セキュリティに関する新しいセクションが作成されました。詳細については、「AWS Device Farm のセキュリティ」を参照してください。	2020 年 3 月 27 日

変更	説明	変更日
	Device Farm でのタグ付けに関する新しいセクションが追加されました。タグ付けの詳細については、「Device Farm でのタギング」を参照してください。	
ダイレクトデバイスアクセスの削除。	直接デバイスアクセス (プライベートデバイスでのリモートデバッグ) は、一般的な使用目的には利用できなくなりました。直接デバイスアクセスの今後の利用可能性については、こちらまでお問い合わせください。	2019 年 9 月 9 日
Gradle プラグイン設定の更新	更新された Gradle プラグイン設定には、カスタマイズ可能なバージョンの Gradle 設定が含まれ、オプションのパラメーターはコメントアウトされています。Device Farm Gradle プラグインのセットアップ の詳細を確認してください。	2019 年 8 月 16 日
XCTest を使用したテストランの新しい要件	XCTest フレームワークを使用するテストランでは、テスト用に構築されたアプリケーションパッケージが Device Farm で必要になります。the section called “XCTest” の詳細を確認してください。	2019 年 2 月 4 日
カスタム環境での Appium Node.js および Appium Ruby テストタイプのサポート	Appium Node.js と Appium Ruby の両方のカスタムテスト環境でテストを実行できます。AWS Device Farm のテストタイプによる作業 の詳細を確認してください。	2019 年 1 月 10 日

変更	説明	変更日
Appium サーバーバージョン 1.7.2 が標準環境とカスタム環境の両方でサポートされるようになりました。カスタムテストスペックの YAML ファイルを使用したバージョン 1.8.1 がカスタムテスト環境でサポートされるようになりました。	Appium サーバーバージョン 1.72、1.71、および 1.6.5 を使用して、標準テスト環境とカスタムのテスト環境の両方でテストを実行できるようになりました。また、カスタムのテスト環境で、カスタムのテストスペック YAML ファイルを使用して、バージョン 1.8.1 および 1.8.0 のテストを実行することもできます。AWS Device Farm のテストタイプによる作業 の詳細を確認してください。	2018 年 10 月 2 日
カスタムのテスト環境	カスタムテスト環境を使用すると、ローカル環境でテストを実行するのと同様にテストの実行ができます。Device Farm では、ライブログやビデオストリーミングのサポートが提供されるようになったため、カスタムのテスト環境で実行されるテストに関するフィードバックを素早く取得できます。カスタムテスト環境による作業 の詳細を確認してください。	2018 年 8 月 16 日
AWS CodePipeline のテストプロバイダとして Device Farm の使用をサポート	リリースプロセスのテストアクションとして AWS Device Farm 実行を使用するAWS CodePipelineように でパイプラインを設定できるようになりました。 CodePipeline を使用すると、リポジトリをビルドステージとテストステージにすばやくリンクして、ニーズに応じてカスタマイズされた継続的な統合システムを実現できます。CodePipeline テストステージでの AWS Device Farm の使用 の詳細を確認してください。	2018 年 7 月 19 日

変更	説明	変更日
プライベートデバイスのサポート	プライベートデバイスを使用してテストランをスケジュールし、リモートアクセスセッションを開始できるようになりました。これらのデバイスのプロファイルと設定の管理、Amazon VPC エンドポイントの作成によるプライベートアプリケーションのテスト、およびリモートデバッグセッションの作成を行うことができます。AWS Device Farm でのプライベートデバイスによる作業 の詳細を確認してください。	2018 年 5 月 2 日
Appium 1.6.3 のサポート	Appium カスタムテストの Appium バージョンを設定できるようになりました。	2017 年 3 月 21 日
テストランの実行タイムアウトを設定する	テストランの実行タイムアウトまたはプロジェクトのすべてのテストの実行タイムアウトを設定できます。AWS Device Farm でのテスト実行の実行タイムアウトを設定する の詳細を確認してください。	2017 年 2 月 9 日
ネットワークシェーピング	テストラン用のネットワーク接続と条件をシミュレートできるようになりました。AWS Device Farm 実行用のネットワークの接続と条件をシミュレートする の詳細を確認してください。	2016 年 12 月 8 日
トラブルシューティングセクションの新規追加	Device Farm コンソールで発生する可能性のあるエラーメッセージを解決するための一連の手順を使用して、テストパッケージのアップロードのトラブルシューティングができるようになりました。Device Farm エラーのトラブルシューティング の詳細を確認してください。	2016 年 8 月 10 日
リモートアクセスセッション	コンソールで単一のデバイスにリモートでアクセスして連携できるようになりました。リモートアクセスによる作業 の詳細を確認してください。	2016 年 4 月 19 日
デバイススロットセルフサービス	AWS Management Console、AWS Command Line Interface、または API を使用して、デバイススロットを購入できるようになりました。Device Farm でデバイススロットを購入する 方法の詳細情報。	2016 年 3 月 22 日

変更	説明	変更日
テストランの停止方法	AWS Management Console、AWS Command Line Interface、または API を使用して、テストランを停止できるようになりました。AWS Device Farm での実行を停止する 方法の詳細情報。	2016 年 3 月 22 日
XCTest UI テストタイプの新規追加	iOS アプリケーションで XCTest UI カスタムテストを実行できるようになりました。XCTest UI テストタイプの詳細情報。	2016 年 3 月 8 日
Appium Python テストタイプの新規追加	Android、iOS、およびウェブアプリケーションで Appium Python カスタムテストを実行できるようになりました。AWS Device Farm のテストタイプによる作業 の詳細を確認してください。	2016 年 1 月 19 日
ウェブアプリケーションテストタイプ	ウェブアプリケーションで Appium Java JUnit および TestNG カスタムテストを実行できるようになりました。AWS Device Farm でのウェブアプリケーションテストによる作業 の詳細を確認してください。	2015 年 11 月 19 日
AWS Device Farm Gradle プラグイン	Device Farm Gradle プラグイン のインストールおよび使用方法の詳細情報。	2015 年 9 月 28 日
Android 組み込みテストの新規追加: エクスプローラー	エクスプローラーテストでは、各画面をエンドユーザーであるかのように分析してアプリケーションをクロールし、調査中のスクリーンショットを取得します。	2015 年 9 月 16 日
iOS サポートの追加	iOS デバイスのテストと iOS テスト (XCTest を含む) の実行の詳細については、「AWS Device Farm のテストタイプによる作業」を参照してください。	2015 年 8 月 4 日
初回一般リリース	これは『AWS Device Farm 開発者ガイド』の初回一般リリースです。	2015 年 7 月 13 日

AWS 用語集

AWS の最新の用語については、「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。