デベロッパーガイド

AWS AppSync GraphQL

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS AppSync GraphQL: デベロッパーガイド

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon の商標およびトレードドレスは、Amazon のものではない製品またはサービスにも関連して、お客様に混乱を招いたり Amazon の信用を傷つけたり失わせたりするいかなる形においても使用することはできません。Amazon が所有していないその他のすべての商標は Amazon との提携、関連、支援関係の有無にかかわらず、それら該当する所有者の資産です。

Table of Contents

とは AWS AppSync 1
AWS AppSync の機能 1
初めての AWS AppSync ユーザーですか？ 2
関連サービス 2
の料金 AWS AppSync 3
GraphQL と AWS AppSync アーキテクチャ 4
API とは 5
クライアント 5
リソース 5
REST とは 6
統一インターフェース 6
ステートレス性 7
階層型システム 7
キャッシュ性 7
RESTful API とは? 7
RESTful API はどのように機能するのでしょうか? 8
GraphQL とは 8
REST と GraphQL の比較 8
REST よりも GraphQL を使用する理由 11
GraphQL API のコンポーネント 12
GraphQL スキーマ 13
データソース 31
リゾルバー 45
GraphQL の追加プロパティ 55
宣言型 55
階層的 56
イントロスペクティブ 57
強力なタイピング 58
はじめに: GraphQL API の作成を開始する 59
スキーマの起動 60
AWS AppSync コンソールのツアー 64
スキーマデザイナー 64
データソース 65
クエリ 66

設定 66
GraphQL ミューテーションを使用して DynamoDB テーブルにデータを追加する 66
GraphQL クエリを使用して DynamoDB テーブルからデータを取得する 72
補足資料セクション 75
Integration 75
補足資料 76
GraphQL API の設計 77
GraphQL API の構築 (空の API またはインポートされた API) 77
GraphQL スキーマの設計 78
データソースを追加する 106
AWS AppSync リゾルバーの設定 118
CDK での API の使用 175
リアルタイムデータアプリケーションのサブスクリプションの使用 193
GraphQL スキーマサブスクリプションディレクティブ 193
サブスクリプション引数の使用 196
サーバーレス WebSockets を利用した汎用的な Pub/sub API の作成 200
拡張サブスクリプションフィルターを定義する 204
フィルターを使用した WebSocket 接続の購読解除 215
リアルタイム WebSocket クライアントの構築 220
API のマージ 241
Merged API とフェデレーション 243
Merged API の競合の解決 244
スキーマの設定 252
承認モードの設定 253
実行ロールの設定 254
AWS RAMを使用したクロスアカウント Merged API の設定 255
マージ 257
Merged API に対するその他のサポート 258
Merged API の制限 259
Merged API の作成 259
RDS イントロスペクションを使用した GraphQL API の構築 261
イントロスペクション機能を使用する (コンソール) 262
イントロスペクション機能の使用 (API) 266
クライアントアプリケーションをビルドする 269
JavaScript リゾルバーチュートリアル 272
DynamoDB JavaScript リゾルバーを使用したシンプルなポストアプリケーションの作成 272

GraphQL API の作成 ... 273
基本的な Post API の定義 ... 273
Amazon DynamoDB テーブルのセットアップ ... 274
addPost リゾルバー (DAmazon DynamoDB PutItem) のセットアップ ... 275
getPost リゾルバー (Amazon DynamoDB GetItem) のセットアップ ... 278
updatePost ミューテーション (Amazon DynamoDB UpdateItem) の作成 ... 281
投票ミューテーションの作成 (Amazon DynamoDB UpdateItem) ... 285
deletePost リゾルバー (Amazon DynamoDB DeleteItem) のセットアップ ... 288
allPost リゾルバー (Amazon DynamoDB Scan) のセットアップ ... 295
「allPostsByAuthor」リゾルバー (Amazon DynamoDB クエリ) のセットアップ ... 299
セット型の使用 ... 304
結論 ... 311
リ AWS Lambda ゾルバーの使用 ... 312
Lambda 関数を作成する ... 312
Lambda のデータソースを設定する ... 313
GraphQL スキーマを作成する ... 315
リゾルバーを設定する ... 120
GraphQL API をテストする ... 316
エラーを返す ... 318
高度なユースケース: バッチ処理 ... 321
ローカルリゾルバーの使用 ... 330
Pub/Sub アプリの作成 ... 330
ページの送信およびサブスクライブ ... 331
GraphQL リゾルバーの組み合わせ ... 333
スキーマの例 ... 333
リゾルバーを使用してデータを変更する ... 334
DynamoDB と OpenSearch Service ... 335
OpenSearch Service リゾルバーの使用 ... 337
新しい OpenSearch Service ドメインを作成する ... 338
OpenSearch Service 用のデータソースの設定 ... 338
リゾルバーを接続する ... 340
検索を変更する ... 342
OpenSearch Service へのデータの追加 ... 343
単一のドキュメントの取得 ... 344
クエリとミューテーションを実行する ... 345
ベストプラクティス ... 346

DynamoDB トランザクションの実行 ... 346
アクセス許可 ... 346
データソース ... 347
トランザクション ... 349
DynamoDB バッチオペレーションの使用 ... 356
1 つのテーブルのバッチ処理 ... 357
複数テーブルのバッチ処理 ... 362
エラー処理 ... 370
HTTP リゾルバーの使用 ... 376
REST API を作成する ... 376
GraphQL API の作成 ... 377
GraphQL スキーマを作成する ... 377
HTTP データソースを設定する ... 378
リゾルバーの設定 ... 152
AWS サービスの呼び出し ... 382
Data API での Aurora PostgreSQL の使用 ... 383
Aurora PostgreSQL データベースをセットアップする ... 384
データベースとテーブルの作成 ... 385
GraphQL スキーマを作成する ... 386
RDS のリゾルバー ... 388
クラスターの削除 ... 397
VTL リゾルバーチュートリアル ... 398
DynamoDB リゾルバーを使用したシンプルなポストアプリケーションの作成 ... 399
DynamoDB テーブルのセットアップ ... 399
GraphQL API の作成 ... 377
基本的な Post API の定義 ... 401
DynamoDB テーブル用のデータソースの設定 ... 402
addPost リゾルバー (DynamoDB PutItem) のセットアップ ... 403
getPost リゾルバー (DynamoDB GetItem) のセットアップ ... 408
updatePost ミューテーション (DynamoDB UpdateItem) の作成 ... 411
updatePost リゾルバー (DynamoDB UpdateItem) の変更 ... 414
upvotePost と downvotePost ミューテーション (DynamoDB UpdateItem) の作成 ... 421
deletePost リゾルバー (DynamoDB DeletePost) のセットアップ ... 425
allPost リゾルバー (DynamoDB Scan) のセットアップ ... 432
「allPostsByAuthor」リゾルバー (DynamoDB クエリ) のセットアップ ... 437
セット型の使用 ... 304

リスト型とマップ型の使用 450
結論 453
リ AWS Lambda ゾルバーの使用 454
Lambda 関数を作成する 454
Lambda のデータソースを設定する 456
GraphQL スキーマを作成する 377
リゾルバーを設定する 152
GraphQL API をテストする 460
エラーを返す 462
高度なユースケース: バッチ処理 465
OpenSearch Service リゾルバーの使用 475
ワンクリックでのセットアップ 475
OpenSearch Service ドメインを作成する 476
OpenSearch Service のデータソースの設定 476
リゾルバーを接続する 478
検索を変更する 480
OpenSearch Service へのデータの追加 481
単一のドキュメントの取得 482
クエリとミューテーションを実行する 483
ベストプラクティス 483
ローカルリゾルバーの使用 484
ページングアプリケーションの作成 484
ページの送信およびサブスクライブ 486
GraphQL リゾルバーの組み合わせ 486
スキーマの例 487
リゾルバーを使用してデータを変更する 488
DynamoDB と OpenSearch Service 489
DynamoDB バッチオペレーションの使用 493
アクセス許可 493
データソース 494
1 つのテーブルのバッチ処理 495
複数テーブルのバッチ処理 499
エラー処理 507
DynamoDB トランザクションの実行 512
アクセス許可 493
データソース 494

トランザクション 515
HTTP リゾルバーの使用 525
ワンクリックでのセットアップ 475
REST API を作成する 376
GraphQL API の作成 377
GraphQL スキーマを作成する 377
HTTP データソースを設定する 378
リゾルバーの設定 152
AWS サービスの呼び出し 531
Aurora Serverless v2 リゾルバーの使用 532
データベースクラスターのセットアップ 534
Data API を有効にする 535
データベースとテーブルを作成する 535
GraphQL スキーマ 536
API をデータベースオペレーションに接続する 152
API を使用してデータを変更する 545
データの取得 546
データアクセスを保護する 547
パイプラインリゾルバーの使用 549
ワンクリックでのセットアップ 475
手動セットアップ 550
GraphQL API をテストする 460
バージョニングされたデータソースでの Delta Sync オペレーションの使用 564
ワンクリックでのセットアップ 475
Schema 565
ミューテーション 568
同期クエリ 568
例 569
構成と設定 576
サーバー側のキャッシュと API ペイロード圧縮の設定 576
インスタンスのタイプ 577
キャッシュの動作 578
キャッシュ暗号化 580
キャッシュエビクション 581
キャッシュエントリーのエビクション 581
ID に基づくキャッシュエントリのエビクション 582

API レスポンスの圧縮 584
GraphQL およびリアルタイム API のカスタムドメイン名の設定 585
ドメイン名を登録および設定する 586
AWS AppSync でカスタムドメイン名を作成する 586
AWS AppSync でのワイルドカードカスタムドメイン名 587
DynamoDB のバージョニング、競合検出、同期オペレーション 588
DynamoDB データソースのバージョニング 589
競合の検出と解決 593
バージョニングされたデータソースでの DynamoDB 同期オペレーションの使用 603
CloudWatch を使用した GraphQL API データのモニタリングとログ 604
セットアップと設定 605
CloudWatch メトリクス 606
CloudWatch ログ 617
ログタイプリファレンス 622
CloudWatch Logs Insights を使用したログの分析 624
OpenSearch Service を使用してログを分析する 626
ログ形式の移行 626
でのリクエストのトレース AWS X-Ray 627
セットアップと設定 605
X-Ray で API をトレースする 628
での AWS CloudTrail API コールのログ記録 630
AWS AppSync CloudTrail の情報 630
AWS AppSync CloudTrail のデータイベント 631
AWS AppSync ログファイルエントリについて 633
AWS AppSync プライベート APIs 640
AWS AppSync プライベート APIsの作成 642
AWS AppSyncのインターフェイスエンドポイントの作成 643
高度な の例 644
IAM ポリシーを使用してパブリック API の作成を制限する 648
GraphQL APIs の共有 649
AWS AppSync GraphQL APIs を共有するための前提条件 649
Share AWS AppSync GraphQL APIs 650
AWS AppSync GraphQL APIs 652
クロスアカウントイベント 653
GraphQL の実行の複雑さ、クエリの深さ、イントロスペクションを設定する 653
イントロスペクション機能を使用する 653

クエリの深さの制限を設定する 655
リゾルバー数の制限の設定 656
環境変数の使用 658
環境変数の設定 (コンソール) 659
環境変数の設定 (API) 659
環境変数の設定 (CFN) 661
環境変数とマージ済みの API 661
環境変数の取得 661
GraphQL API の認可と認証 663
認可タイプ 663
API_KEY 認可 664
AWS_LAMBDA 認可 666
SigV4 と OIDC トークンの認可制限を回避する 672
AWS_IAM 認可 672
OPENID_CONNECT 認可 675
AMAZON_COGNITO_USER_POOLS 認可 676
追加の承認モードの使用 678
きめ細かなアクセスコントロール 680
フィルタ処理情報 683
データソースへのアクセス 684
リクエストとレスポンスを保護するためのアクセスコントロールのユースケース 684
概要 685
データの読み込み 685
データの書き込み 689
パブリックレコードとプライベートレコード 692
リアルタイムデータ 693
AWS WAF を使用して APIsを保護する 696
AppSync API を と統合する AWS WAF 697
ウェブ ACL のルールの作成 699
セキュリティ 703
データ保護 704
転送中の暗号化 705
コンプライアンス検証 705
インフラストラクチャセキュリティ 706
耐障害性 707
Identity and Access Management 707

対象者 ... 708
アイデンティティを使用した認証 ... 708
ポリシーを使用したアクセスの管理 ... 712
IAM での AWS AppSync の仕組み ... 715
アイデンティティベースのポリシー ... 721
トラブルシューティング ... 734
を使用した Logging AWS AppSync API コール AWS CloudTrail ... 737
CloudTrail のAWS AppSync 情報 ... 737
AWS AppSync ログファイルエントリについて ... 738
ベストプラクティス ... 483
認証方法を理解する ... 740
API 設定の変更がどのように伝達されるかを理解する ... 741
HTTP リゾルバーに TLS を使用する ... 741
最小の権限を持つロールを使用する ... 741
IAM ポリシーのベストプラクティス ... 741
リゾルバーリファレンス (JavaScript) ... 743
JavaScript リゾルバーの概要 ... 743
サポートされているランタイム機能 ... 744
ユニットリゾルバー ... 744
JavaScript パイプラインリゾルバーの仕組み ... 744
補足トピック ... 749
Amazon DynamoDB を使用したパイプラインリゾルバーの例 ... 749
APPSYNC_JS ランタイムのユーティリティの設定 ... 752
APPSYNC_JS ランタイムのバンドル、TypeScript、ソースマップ ... 755
リゾルバーと関数ハンドラーのテスト ... 761
VTL から JavaScript への移行 ... 764
データソースへの直接アクセスと Lambda データソース経由のプロキシのどちらかを選択する ... 767
JavaScript リゾルバーコンテキストオブジェクトリファレンス ... 769
context へのアクセス ... 769
リゾルバーおよび関数の JavaScript runtime 機能 ... 780
サポートされているランタイム機能 ... 781
組み込みユーティリティ ... 788
ビルトインモジュール ... 791
ランタイムユーティリティ ... 815
util.time の日時ヘルパー ... 816

util.dynamodb の DynamoDB ヘルパー 817
util.http の HTTP ヘルパー 823
util.transform の変換ヘルパー 825
util.str の文字列ヘルパー 839
拡張子 839
util.xml の XML ヘルパー 843
DynamoDB 用の JavaScript リゾルバー関数リファレンス 845
GetItem 847
PutItem 849
UpdateItem 852
DeleteItem 857
クエリ 859
Scan 864
同期 868
BatchGetItem 871
BatchDeleteItem 874
BatchPutItem 876
TransactGetItems 879
TransactWriteItems 882
型システム (リクエストマッピング) 888
型システム (レスポンスマッピング) 893
フィルター 897
条件式 899
トランザクション条件式 912
計画 915
OpenSearch 用 JavaScript リゾルバー関数リファレンス 916
リクエスト 917
レスポンス 917
`operation` フィールド 918
`path` フィールド 918
`params` フィールド 919
渡す変数 921
Lambda 用の JavaScript リゾルバー関数リファレンス 922
オブジェクトをリクエストする 922
レスポンスオブジェクト 926
Lambda 関数のバッチ処理されたレスポンス 926

EventBridge データソース用 JavaScript リゾルバー関数リファレンス 926
リクエスト 917
レスポンス 927
PutEvents フィールド 929
None データソース用 JavaScript リゾルバー関数リファレンス 930
リクエスト 917
ペイロード 924
レスポンス 927
HTTP JavaScript 対応のリゾルバー関数リファレンス 932
リクエスト 917
[メソッド] 932
ResourcePath 933
Params フィールド 933
レスポンス 927
Amazon RDS の JavaScript リゾルバー関数リファレンス 935
SQL タグ付きテンプレート 935
ステートメントの作成 936
データの取得 937
ユーティリティ関数 938
キャスティング 946
Amazon Bedrock の JavaScript リゾルバー関数リファレンス 947
オブジェクトをリクエストする 948
レスポンスオブジェクト 955
長時間実行される呼び出し 955
タイプリファレンス 956
リゾルバーのマッピングテンプレートリファレンス (VTL) 963
リゾルバーのマッピングテンプレートの概要 964
ユニットリゾルバー 964
パイプラインリゾルバー 171
サンプルテンプレート 969
評価されたマッピングテンプレートの逆シリアル化ルール 971
リゾルバーのマッピングテンプレートプログラミングガイド 973
セットアップ 974
[変数] 975
メソッドの呼び出し 977
文字列 978

loop 979
配列 980
条件チェック 980
演算子 981
Context 983
フィルタリング 983
リゾルバーのマッピングテンプレートのコンテキストリファレンス 989
$context へのアクセス 989
入力のサニタイズ 999
リゾルバーのマッピングテンプレートユーティリティーリファレンス 1000
$util のユーティリティヘルパー 1001
AWS AppSync ディレクティブ 1014
$util.time の日時ヘルパー 1014
$util.list のヘルパーのリスト 1017
$util.map のマップヘルパー 1018
$util.dynamodb の DynamoDB ヘルパー 1018
$util.rds の Amazon RDS ヘルパー 1028
$util.http の HTTP ヘルパー 1031
$util.xml の XML ヘルパー 1034
util.transform の変換ヘルパー 1036
$util.math の math ヘルパー 1049
$util.str 内の文字列ヘルパー 1050
拡張子 1051
DynamoDB のリゾルバーのマッピングテンプレートリファレンス 1065
GetItem 1067
PutItem 1069
UpdateItem 1072
DeleteItem 1078
クエリ 1081
Scan 1086
同期 1090
BatchGetItem 1094
BatchDeleteItem 1098
BatchPutItem 1101
TransactGetItems 1105
TransactWriteItems 1109

型システム (リクエストマッピング) 1117
型システム (レスポンスマッピング) 1122
フィルター 1126
条件式 1128
トランザクション条件式 1141
計画 1144
RDS のリゾルバーのマッピングテンプレートリファレンス 1145
リクエストマッピングテンプレート 1146
バージョン 1147
statements と VariableMap 1148
VariableTypeHintMap 1148
OpenSearch 用リゾルバーのマッピングテンプレートリファレンス 1149
リクエストマッピングテンプレート 1146
レスポンスマッピングテンプレート 917
operation フィールド 918
path フィールド 918
params フィールド 919
渡す変数 921
Lambda のリゾルバーのマッピングテンプレートリファレンス 1154
リクエストマッピングテンプレート 1146
レスポンスマッピングテンプレート 917
Lambda 関数のバッチ処理されたレスポンス 1160
ダイレクトLambda リゾルバー 1160
EventBridge のリゾルバーのマッピングテンプレートリファレンス 1166
リクエストマッピングテンプレート 1146
レスポンスマッピングテンプレート 917
PutEvents フィールド 929
None データソース用リゾルバーマッピングテンプレートリファレンス 1171
リクエストマッピングテンプレート 1146
バージョン 1147
ペイロード 1158
レスポンスマッピングテンプレート 917
HTTP 用リゾルバーのマッピングテンプレートリファレンス 1173
リクエストマッピングテンプレート 1146
バージョン 1147
[メソッド] 1176

ResourcePath 1176
Params フィールド 919
レスポンス 1178
HTTPS エンドポイント AWS AppSync で によって認識される認証機関 (CA) 1179
リゾルバマッピングテンプレートの変更ログ 1241
バージョンごとのデータソースオペレーションの可用性一覧 1242
ユニットリゾルバーのマッピングテンプレートのバージョンの変更 1243
関数のバージョンの変更 1244
2018-05-29 1244
2017-02-28 1251
GraphQL タイプのリファレンス 1252
GraphQL のスカラー型 1253
デフォルトスカラー 1253
AWS AppSync スカラー 1254
スキーマの使用例 1255
GraphQL の Interface と Union 1258
インターフェースの例 1259
ユニオンの例 1263
AWS AppSync でのタイプ解決 1264
タイプ解決の例 1264
トラブルシューティングと一般的な誤り 1270
DynamoDB キーのマッピングが正しくない 1270
リゾルバーがない 1270
マッピングテンプレートのエラー 1271
戻り値の型が正しくない 1271
無効なリクエストの処理 1272
..... mcclxxiii

とは AWS AppSync

AWS AppSync を使用すると、開発者は安全でサーバーレスで高性能な GraphQL および Pub/Sub APIs を使用して、アプリケーションとサービスをデータやイベントに接続できます。 AWS AppSync を使用して以下を実行できます。

- 1 つの GraphQL API エンドポイントから 1 つ以上のデータソースのデータにアクセスする。
- 複数のソース GraphQL API を組み合わせて 1 つのマージされた GraphQL API にする。
- リアルタイムのデータ更新をアプリケーションにパブリッシュします。
- 組み込みのセキュリティ、モニタリング、ロギング、トレーシングを活用し、オプションのキャッシュによりレイテンシーを低く抑えます。
- API リクエストと配信されたリアルタイムメッセージに対してのみ料金が発生します。

> **Important**
>
> 2025 年 3 月 13 日現在、 AWS AppSync Events を使用して WebSockets を搭載したリアルタイム PubSub API を構築できます。詳細については、AppSync Events デベロッパーガイドのWebSocket 経由でイベントを発行する」を参照してください。 AWS AppSync

トピック

- AWS AppSync の機能
- 初めての AWS AppSync ユーザーですか？
- 関連サービス
- の料金 AWS AppSync

AWS AppSync の機能

- GraphQL によるシンプルなデータアクセスとクエリ
- GraphQL サブスクリプションとパブ/サブチャンネル用のサーバーレス WebSockets
- サーバー側のキャッシュにより、高速のインメモリキャッシュでデータを利用できるようになるため、レイテンシーが低くなります。

- ビジネスロジックを書くための JavaScript とタイプスクリプトのサポート
- API へのアクセスと AWS WAFとの統合を制限するプライベート API によるエンタープライズセキュリティ
- API キー、IAM、Amazon Cognito、OpenID Connect プロバイダー、カスタムロジック用の Lambda 認可をサポートする組み込みの認可コントロールを備えています。
- フェデレーションされたユースケースをサポートする Merged API

これらの各機能の詳細については、「AWS AppSyncの機能」を参照してください。

初めての AWS AppSync ユーザーですか？

初回の AWS AppSync ユーザーは、以下のセクションを読むことから始めることをお勧めします。

- GraphQL に慣れていない場合は、「はじめに: GraphQL API の作成を開始する」を参照してください。
- GraphQL API を使用するアプリケーションを構築する場合は、「クライアントアプリケーションをビルドする」および「the section called “リアルタイムデータアプリケーションのサブスクリプションの使用”」を参照してください。
- GraphQL リゾルバーの情報については、以下を参照してください。

  JavaScript/TypeScript
  - リゾルバーチュートリアル (JavaScript)
  - リゾルバーリファレンス (JavaScript)

  VTL
  - リゾルバーチュートリアル (VTL)
  - リゾルバーのマッピングテンプレートリファレンス (VTL)
- サンプル AWS AppSync プロジェクト、更新などをお探しの場合は、AppSync ブログを参照してください。

関連サービス

ウェブアプリやモバイルアプリをゼロから構築する場合は、AWS Amplify の使用を検討してください。Amplify は AWS AppSync およびその他の AWS のサービスを活用して、より少ない作業でより堅牢で強力なウェブおよびモバイルアプリを構築するのに役立ちます。

の料金 AWS AppSync

AWS AppSync は、数百万のリクエストと更新に基づいて料金が設定されます。キャッシュには追加料金がかかります。詳細については、「AWS AppSync 料金表」を参照してください。

一般 AWS AppSync 料金の例外を以下に示します。

- 認可および認証の失敗でリクエストに課金されることはありません。
- API キーを必要とするメソッドを呼び出す場合、API キーが不足しているまたは無効であれば、課金されません。

GraphQL と AWS AppSync アーキテクチャ

> **Note**
>
> このガイドは、ユーザーが REST アーキテクチャスタイルに関する実用的な知識を持っていることを前提としています。GraphQL および AWS AppSyncを使用する前に、このトピックとその他のフロントエンドトピックを確認することをお勧めします。

GraphQL は API 用のクエリおよび操作言語です。GraphQL は、データ要件と相互作用を記述するための柔軟で直感的な構文を提供します。これにより、開発者は必要なものを正確に尋ねて、予測可能な結果を得ることができます。また、1 回のリクエストで多数のソースにアクセスできるようになるため、ネットワーク呼び出し回数と帯域幅要件が減り、バッテリー寿命と、アプリケーションが消費する CPU サイクルを節約できます。

データの更新はミューテーションによって簡単に行えるため、開発者はデータをどのように変更すべきかを説明できます。また、GraphQL はサブスクリプションによるリアルタイムソリューションの Quick Setup も容易にします。これらすべての機能と強力な開発者ツールを組み合わせることで、GraphQL はアプリケーションデータの管理に欠かせないものになっています。

GraphQL は REST に代わるものです。RESTful アーキテクチャは、現在、クライアントとサーバー間の通信において最もポピュラーなソリューションの 1 つです。リソース (データ) が URL によって公開されるという概念が中心になっています。これらの URL を使用すると GET、POST、DELETE などの HTTP メソッド形式の CRUD (作成、読み取り、更新、削除) 操作を通じてデータにアクセスし、データを操作できます。REST の利点は、学習と実装が比較的簡単であることです。RESTful API を素早く設定して、さまざまなサービスを呼び出すことができます。

しかし、テクノロジーはますます複雑になっています。アプリケーション、ツール、サービスが世界中のユーザー向けに拡大し始める中、高速でスケーラブルなアーキテクチャの必要性が最も重要になっています。スケーラブルな運用を扱う場合、REST には多くの欠点があります。例としてこのユースケース を参照してください。

以下のセクションでは、RESTful API に関するいくつかの概念を確認します。次に、GraphQL とその仕組みを紹介します。

GraphQL の詳細と への移行の利点については AWS、GraphQL 実装に関する決定ガイド」を参照してください。

トピック

- API とは何ですか?
- REST とは?
- GraphQL とは
- REST と GraphQL の比較
- なぜ REST よりも GraphQL を使うのか ?
- GraphQL API のコンポーネント
- GraphQL の追加プロパティ

API とは何ですか?

アプリケーションプログラミングインターフェイス (API) は、他のソフトウェアシステムと通信するために従わなければならないルールを定義します。開発者は API を公開または作成して、他のアプリケーションがプログラムによってアプリケーションと通信できるようにします。例えば、タイムシートアプリケーションでは、従業員のフルネームと日付範囲を尋ねる API を公開しています。この情報を受け取ると、従業員のタイムシートを内部で処理し、その日付範囲内の労働時間数を返します。

Web API は、クライアントとウェブ上のリソース間のゲートウェイと考えることができます。

クライアント

クライアントはウェブ上の情報にアクセスしたいユーザーです。クライアントは、個人でも、API を使用するソフトウェアシステムでもかまいません。例えば、開発者は気象システムから気象データにアクセスするプログラムを作成できます。また、天気 Web サイトに直接アクセスしたときに、ブラウザから同じデータにアクセスすることもできます。

リソース

リソースは、さまざまなアプリケーションがクライアントに提供する情報です。リソースには、画像、動画、テキスト、数字、またはあらゆる種類のデータがあります。リソースをクライアントに提供するマシンはサーバーとも呼ばれます。組織は API を使用してリソースを共有し、セキュリティ、制御、認証を維持しながら Web サービスを提供します。さらに、API は、どのクライアントが特定の内部リソースにアクセスできるかを判断するのに役立ちます。

REST とは?

Representational State Transfer (REST) は、大まかに言うと API の動作に条件を課すソフトウェアアーキテクチャです。REST は当初、インターネットのような複雑なネットワーク上の通信を管理するためのガイドラインとして作成されました。REST ベースのアーキテクチャを使用すると、高性能で信頼性の高い通信を大規模にサポートできます。実装や変更が容易なため、あらゆる API システムの可視性とクロスプラットフォームの移植性を実現できます。

API 開発者は、複数の異なるアーキテクチャを使用して API を設計できます。REST アーキテクチャスタイルに従う API は REST API と呼ばれます。REST アーキテクチャを実装する Web サービスは RESTful Web サービスと呼ばれます。RESTful API という用語は、一般的に RESTful ウェブ API を指します。ただし、REST API と RESTful API という用語は同じ意味で使用できます。

REST アーキテクチャスタイルの原則は以下のとおりです。

統一インターフェース

統一されたインターフェースは、あらゆる RESTful Web サービスの設計の基本です。これは、サーバーが情報を標準形式で転送することを示しています。フォーマットされたリソースは REST では表現と呼ばれます。この形式は、サーバーアプリケーション上のリソースの内部表現とは異なる場合があります。例えば、サーバーはデータをテキストとして保存し、HTML 表現形式で送信できます。

統一インターフェースにはアーキテクチャ上の制約が 4 つあります。

1. リクエストではリソースを特定する必要があります。そのためには、統一されたリソース識別子を使用します。
2. クライアントはリソース表現に十分な情報を持っており、必要に応じてリソースを変更または削除できます。サーバーは、リソースを詳しく説明するメタデータを送信することで、この条件を満たします。
3. クライアントは、表現をさらに処理する方法に関する情報を受け取ります。サーバーは、クライアントがそれらを最適に使用する方法に関するメタデータを含むわかりやすいメッセージを送信することでこれを実現しています。
4. クライアントは、タスクを完了するのに必要なその他すべての関連リソースに関する情報を受け取ります。サーバーは、クライアントがより多くのリソースを動的に見つけられるように、表現にハイパーリンクを送信することでこれを実現しています。

ステートレス性

REST アーキテクチャーにおけるステートレス性とは、サーバーが以前のすべての要求とは無関係にすべてのクライアント要求を完了させる通信方法を指します。クライアントは任意の順序でリソースをリクエストでき、すべてのリクエストはステートレスであるか、他のリクエストから分離されます。この REST API の設計上の制約は、サーバーが毎回リクエストを完全に理解して処理できることを意味します。

階層型システム

階層型システムアーキテクチャでは、クライアントはクライアントとサーバー間の他の許可された仲介者に接続でき、それでもサーバーからの応答を受信します。サーバーは他のサーバーに要求を渡すこともできます。RESTful Web サービスは、セキュリティ、アプリケーション、ビジネスロジックなどの複数のレイヤーを持つ複数のサーバー上で動作し、連携してクライアントのリクエストに応えるように設計できます。これらの階層はクライアントには見えないままです。

キャッシュ性

RESTful な Web サービスはキャッシュをサポートします。キャッシュとは、サーバーの応答時間を短縮するために、一部の応答をクライアントまたは仲介者に保存するプロセスです。例えば、すべてのページに共通のヘッダーとフッターの画像がある Web サイトにアクセスしたとします。新しい Web サイトページにアクセスするたびに、サーバーは同じ画像を再送信する必要があります。これを避けるため、クライアントは最初の応答後にこれらの画像をキャッシュまたは保存し、キャッシュから直接画像を使用します。RESTful な Web サービスは、自身をキャッシュ可能またはキャッシュ不可と定義する API レスポンスを使用してキャッシュを制御します。

RESTful API とは?

RESTful API は、2 つのコンピューターシステムがインターネット上で安全に情報を交換するために使用するインターフェースです。ほとんどのビジネスアプリケーションは、さまざまなタスクを実行するために他の内部アプリケーションやサードパーティアプリケーションと通信する必要があります。例えば、毎月の給与明細を生成するには、内部会計システムが顧客の銀行システムとデータを共有して請求を自動化し、社内のタイムシートアプリケーションと通信する必要があります。RESTful API は、安全で信頼性が高く、効率的なソフトウェア通信標準に従っているため、このような情報交換をサポートします。

RESTful API はどのように機能するのでしょうか?

RESTful API の基本的な機能は、インターネットを閲覧することと同じです。リソースが必要な場合、クライアントは API を使用してサーバーに接続します。API 開発者は、クライアントが REST API をどのように使用すべきかをサーバーアプリケーション API ドキュメントで説明しています。REST API コールの一般的な手順は次のとおりです。

1. クライアントはリクエストをサーバーに送信します。クライアントは API ドキュメントに従い、サーバーが理解できる方法でリクエストをフォーマットします。
2. サーバーはクライアントを認証し、クライアントにそのリクエストを行う権利があることを確認します。
3. サーバーはリクエストを受け取り、内部で処理します。
4. サーバーは、クライアントに対してレスポンスを返します。レスポンスには、リクエストが成功したかどうかをクライアントに伝える情報が含まれます。レスポンスには、クライアントがリクエストしたすべての情報も含まれます。

REST API のリクエストとレスポンスの詳細は、API 開発者が API をどのように設計したかによって若干異なります。

GraphQL とは

GraphQL は API のクエリ言語であり、それらのクエリを実行するランタイムでもあります。GraphQL を使用すると、クライアントは必要なデータを正確にリクエストできるため、多くのシナリオで REST よりも柔軟で効率的な代替手段となります。事前定義されたエンドポイントに依存する REST とは異なり、GraphQL は 1 つのエンドポイントを使用し、クライアントはクエリとミューテーションの形式でデータ要件を指定できます。

GraphQL API の構造の詳細については、「GraphQL API のコンポーネント」を参照してください。

REST と GraphQL の比較

API (アプリケーションプログラミングインターフェイス) は、アプリケーション間のデータ交換を容易にする上で重要な役割を果たします。前述のように、API を設計するための GraphQL と REST という 2 つの顕著なアプローチが登場しています。どちらもクライアントとサーバーの通信を可能にする基本的な目的を果たしますが、実装とユースケースでは大きく異なります。

GraphQL と REST には、いくつかの重要な特徴があります。

1. クライアント/サーバーモデル: どちらもデータ交換にクライアント/サーバーアーキテクチャを使用します。
2. ステートレス: どちらもリクエスト間でクライアントセッション情報を維持しません。
3. HTTP ベース: どちらも通常、基本となる通信プロトコルとして HTTP を使用します。
4. リソース指向設計: どちらも、クライアントが API を介してアクセスおよび操作できるデータまたはオブジェクトを参照するリソースに関するデータ交換を設計します。
5. データ形式の柔軟性: 両方で最も一般的に使用されるデータ交換形式は JSON ですが、XML や HTML などの他の形式もサポートされています。
6. 言語とデータベースに依存しない: どちらも任意のプログラミング言語またはデータベース構造で動作するため、高い相互運用性があります。
7. キャッシュサポート: どちらもキャッシュをサポートしているため、クライアントとサーバーは頻繁にアクセスされるデータを保存してパフォーマンスを向上させることができます。

GraphQL と REST は、いくつかの基本原則を共有しながら、API 設計とデータフェッチに対するアプローチが大きく異なります。

1. リクエスト構造とデータ取得

   REST は、さまざまな HTTP メソッド (GET、POST、PUT、DELETE) を使用して、リソースに対してオペレーションを実行します。これには、多くの場合、異なるリソースに複数のエンドポイントが必要になるため、データの取得が非効率になる可能性があります。例えば、GET オペレーションを実行してユーザーのデータを取得すると、データのオーバーフェッチまたはアンダーフェッチが発生する可能性があります。正しいデータを取得するには、切り捨てまたは複数のオペレーションを呼び出すことができます。

   GraphQL は、すべてのオペレーションに単一のエンドポイントを使用します。データの取得にはクエリに依存し、データの変更にはミューテーションに依存します。クライアントはクエリを使用して、1 回のリクエストで必要なデータを正確に取得できるため、データ転送を最小限に抑えることでネットワークオーバーヘッドを削減できます。

2. サーバー側のスキーマ

   REST にはサーバー側のスキーマは必要ありませんが、効率的な API 設計とドキュメント化のためにオプションで定義できます。

   GraphQL は、厳密に型指定されたサーバー側のスキーマを使用して、データとデータサービスを定義します。GraphQL Schema Definition Language (SDL) で記述されたスキーマには、各オブ

ジェクトのオブジェクトタイプとフィールド、および各フィールドのオペレーションを定義するサーバー側のリゾルバー関数が含まれます。

3. バージョニング

   REST には URL にバージョニングが含まれていることがよくあり、複数の API バージョンを同時に維持できます。バージョニングは必須ではありませんが、変更の中断を防ぐのに役立ちます。

   GraphQL は、下位互換性を必要とすることで、明示的なバージョニングなしで API の継続的な進化を促進します。削除されたフィールドはエラーメッセージを返し、非推奨タグは古いフィールドを段階的に廃止し、警告メッセージを返します。

4. エラー処理

   REST はタイプが弱いため、エラー処理を周囲のコードに組み込む必要があります。これにより、タイプ関連のエラー (例: 数値をテキストとして解析) が自動的に識別されない場合があります。

   対照的に、GraphQL は強く入力され、包括的なスキーマ定義が必要です。これにより、サービスが多くのリクエストエラーを高い詳細レベルで自動的に識別できるようになります。

5. ユースケース

   REST は次の用途に適しています。

   - データ要件がそれほど複雑ではない小規模なアプリケーション。
   - データおよびオペレーションがすべてのクライアントで同様に使用されるシナリオ。
   - 複雑なデータクエリのニーズがないアプリケーション。

   GraphQL は次の用途に適しています。

   - リクエストとレスポンスを最小限に抑えることが重要な、帯域幅が限られているシナリオ。
   - 単一のエンドポイントで組み合わせる必要がある複数のデータソースを持つアプリケーション。
   - クライアントリクエストが大きく異なり、異なるレスポンス構造を期待する場合。

   GraphQL API と REST API の両方を 1 つのアプリケーション内で、さまざまな機能分野に使用できることに注意してください。さらに、完全な書き換えなしで GraphQL 機能を含めるように RESTful API をアップグレードできます。例については、 AWS 「データソースの GraphQL リゾルバーを構築する方法」を参照してください。

なぜ REST よりも GraphQL を使うのか？

REST はウェブ API の基本的なアーキテクチャスタイルの 1 つです。しかし、世界の相互接続が進むにつれて、堅牢でスケーラブルなアプリケーションを開発する必要性がより喫緊の課題となるでしょう。REST は現在、Web API を構築するための業界標準ですが、RESTful な実装には繰り返し発生するいくつかの欠点があることが確認されています。

1. データリクエスト: RESTful API を使用する場合、通常はエンドポイントを通じて必要なデータをリクエストします。この問題は、データがきちんとパッケージ化されていない場合に発生します。必要なデータが複数の抽象レイヤーに隠れている場合があり、データを取得する唯一の方法は複数のエンドポイントを使用することです。つまり、すべてのデータを抽出するために複数のリクエストを行うということです。
2. オーバーフェッチとアンダーフェッチ: 複数のリクエストの問題に加えて、各エンドポイントからのデータは厳密に定義されています。つまり、技術的に必要でなくても、その API に定義されているデータはすべて返されるということです。

   その結果、オーバーフェッチが発生し、リクエストから余分なデータが返されることになります。例えば、会社の人事データをリクエストしていて、特定の部門の従業員の名前を知りたいとします。データを返すエンドポイントには名前が含まれますが、役職や生年月日などの他のデータも含まれる場合があります。API は固定されているため、名前だけをリクエストすることはできません。残りのデータはそれに付随します。

   これとは逆に、十分なデータが返されない状況をアンダーフェッチと呼びます。リクエストされたデータをすべて取得するには、サービスに対して複数のリクエストを行う必要がある場合があります。データの構造によっては、非効率的なクエリに遭遇し、恐ろしい n+1 問題のような問題が発生する可能性があります。
3. 開発の反復に時間がかかる: 多くの開発者は、アプリケーションのフローに合わせて RESTful API を調整しています。ただし、アプリケーションが大きくなるにつれて、フロントエンドとバックエンドの両方に大幅な変更が必要になる可能性があります。その結果、API はもはや効率的でも影響力のある形でもデータの形状に適合しなくなる可能性があります。その結果、API を変更する必要があるため、製品のイテレーションが遅くなります。
4. 大規模環境でのパフォーマンス: こうした複合的な問題により、スケーラビリティが影響を受ける分野は多数あります。リクエストから返されるデータが多すぎる、または少なすぎる (結果的にリクエストが増える) ため、アプリケーション側のパフォーマンスが影響を受ける可能性があります。どちらの状況でも、ネットワークに不必要な負荷がかかり、パフォーマンスが低下します。開発者側では、API が固定され、リクエストしているデータに適合しなくなるため、開発速度が低下する可能性があります。

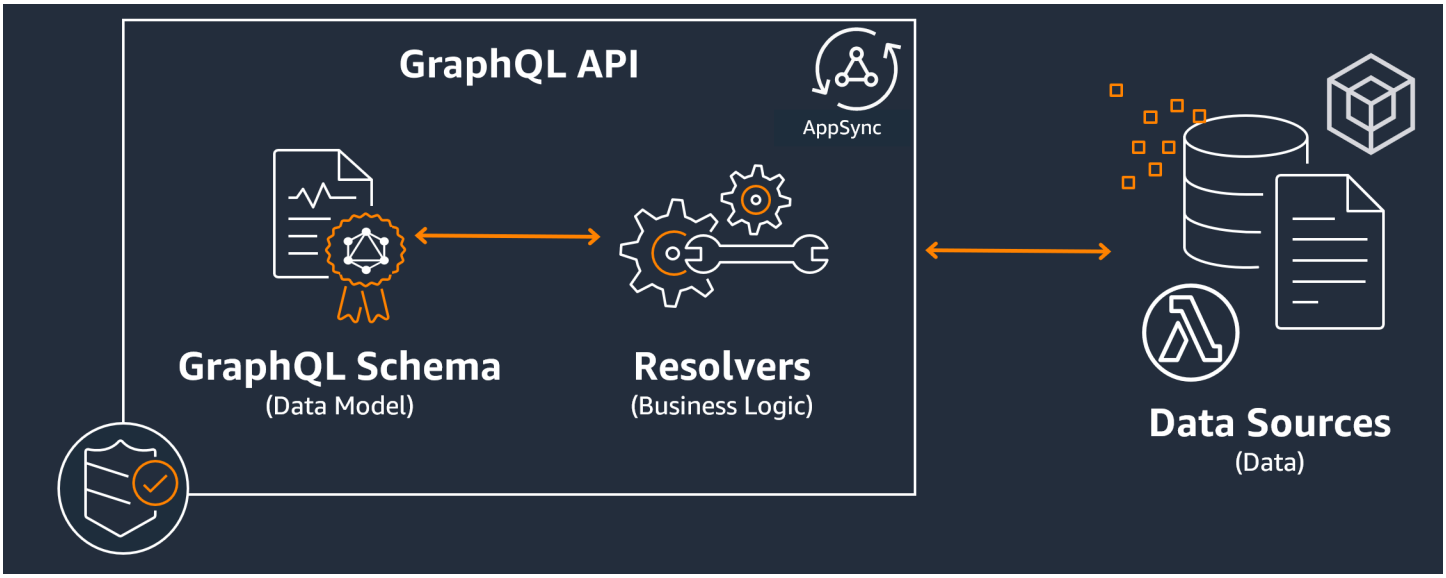
GraphQL のセールスポイントは、REST の欠点を克服することです。GraphQL が開発者に提供する主なソリューションは以下のとおりです。

1. 単一エンドポイント: GraphQL は単一のエンドポイントを使用してデータをクエリします。データの形状に合わせて複数の API を構築する必要はありません。その結果、ネットワークを経由するリクエストが少なくなります。
2. フェッチ: GraphQL は、必要なデータを定義するだけで、オーバーフェッチとアンダーフェッチという長年の問題を解決します。GraphQL では、ニーズに合わせてデータを形作ることができるので、求めたものだけを受け取ることができます。
3. 抽象化: GraphQL API には、言語にとらわれない標準を使用してデータを記述するいくつかのコンポーネントとシステムが含まれています。つまり、データの形状と構造が標準化されているので、フロントエンドとバックエンドの両方がネットワーク上でどのように送信されるかがわかります。これにより、開発者は両方とも GraphQL のシステムを操作でき、そのまわりに煩わされることはありません。
4. 迅速な反復: データの標準化により、開発の一方の端での変更が他方の端では必要ない場合があります。例えば、GraphQL ではデータ仕様を簡単に変更できるため、フロントエンドのプレゼンテーションを変更しても、バックエンドに大きな変更が加えられない場合があります。アプリケーションの成長に合わせて、データの形状を定義または変更するだけで、アプリケーションのニーズに合わせることができます。その結果、潜在的な開発作業が少なくなります。

これらは、GraphQL の利点のほんの一部です。次のいくつかのセクションでは、GraphQL の構造と、GraphQL を REST のユニークな代替手段にするプロパティについて学びます。

GraphQL API のコンポーネント

標準の GraphQL API は、クエリされるデータの形状を処理する単一のスキーマで構成されています。スキーマは、データベースや Lambda 関数などの 1 つ以上のデータソースにリンクされています。2 つのリゾルバーの間には、リクエストのビジネスロジックを処理する 1 つ以上のリゾルバーがあります。各コンポーネントは、GraphQL 実装において重要な役割を担います。以下のセクションでは、これら 3 つのコンポーネントと、それらが GraphQL サービスで果たす役割を紹介します。



トピック

- GraphQL スキーマ
- データソース
- リゾルバー

GraphQL スキーマ

GraphQL スキーマは GraphQL API の基盤です。データの形状を定義する設計図として機能します。また、データの取得方法や変更方法を定義する、クライアントとサーバー間の契約でもあります。

GraphQL スキーマは スキーマ定義言語 (SDL) で記述されています。SDL は、構造が確立された型とフィールドで構成されています。

- タイプ: タイプとは、GraphQL がデータの形状と動作を定義する方法です。GraphQL は、このセクションの後半で説明する多数の型をサポートしています。スキーマで定義されている各タイプには、独自のスコープが含まれます。スコープ内には、GraphQL サービスで使用される値または口ジックを含むことができる 1 つ以上のフィールドがあります。型にはさまざまな役割がありますが、最も一般的なのはオブジェクトまたはスカラー (プリミティブ値型) です。
- フィールド: フィールドはタイプのスコープ内に存在し、GraphQL サービスから要求された値を保持します。これらは他のプログラミング言語の変数とよく似ています。フィールドで定義するデータの形状によって、リクエスト/レスポンス操作におけるデータの構造が決まります。これにより、開発者はサービスのバックエンドがどのように実装されているかを知らなくても、何が返されるかを予測できます。

スキーマがどのようなものかを視覚化するために、単純な GraphQL スキーマの内容を確認してみましょう。プロダクションコードでは、スキーマは通常、schema.graphql または schema.json というファイルにあります。GraphQL サービスを実装するプロジェクトを覗き見していると仮定しましょう。このプロジェクトには会社の人事データが保存されており、schema.graphql ファイルを使用して人事データを取得したり、データベースに新しい人員を追加したりしています。コードは次のようになります。

schema.graphql

```
type Person {
   id: ID!
   name: String
   age: Int
}
type Query {
  people: [Person]
}
type Mutation {
  addPerson(id: ID!, name: String, age: Int): Person
}
```

スキーマには、Person、Query、Mutation の 3 つのタイプが定義されていることがわかります。Person を見てみると、これが会社の従業員のインスタンスの設計図であり、それによってこの型がオブジェクトになることが推測できます。そのスコープ内には、id、name、age があります。これらは Person のプロパティを定義するフィールドです。つまり、データソースはそれぞれのPersonのnameをStringスカラー (プリミティブ) タイプとして、age をスカラー (プリミティブ) タイプとして格納します。id はそれぞれの Person に対して固有の特別な識別子として機能します。! 記号で示されているように、これは必須の値でもあります。

次の 2 つのオブジェクトタイプは動作が異なります。GraphQL は、スキーマへのデータの入力方法を定義する特別なオブジェクトタイプ用にいくつかのキーワードを予約しています。Query タイプはソースからデータを取得します。この例では、クエリはデータベースから Person オブジェクトを取得する場合があります。これは、RESTful 用語で使われる GET 操作を思い起こさせるかもしれません。Mutation はデータを変更します. この例では、ミューテーションによってデータベースにさらに Person オブジェクトが追加される可能性があります。これにより、PUT や POST のような状態を変更する操作を思い起こさせるかもしれません。特殊なオブジェクトタイプの動作については、このセクションの後半で説明します。

この例では、Query がデータベースから何かを取得すると仮定しましょう。Query のフィールドを見ると、people というフィールドが 1 つあります。フィールドの値は [Person] です。つまり、データベース内の Person の一部のインスタンスを取得したいということです。ただし、括弧を追加すると、特定のインスタンスだけでなく、すべての Person インスタンスのリストを返したいということになります。

データ変更などの状態を変更する操作は Mutation タイプが担当します。ミューテーションは、データソースに対して何らかの状態変更操作を実行する役割を果たします。この例では、データベースに新しい addPerson オブジェクトを追加する Person という操作がミューテーションに含まれています。このミューテーションは Person を使用し、id、name、age フィールドへの入力を期待しています。

この時点で、このような操作は何らかの動作を行い、関数名とパラメータを持つ関数によく似ているはずなのに、コード実装なしで addPerson がどのように動作するのか疑問に思われるかもしれません。現在のところ、スキーマは宣言の役割を果たすだけなので、機能しません。addPerson の動作を実装するには、リゾルバーを追加する必要があります。リゾルバーは、関連するフィールド (この場合は addPerson オペレーション) が呼び出されるたびに実行されるコードの単位です。オペレーションを使いたい場合は、どこかの時点でリゾルバーの実装を追加する必要があります。ある意味では、スキーマ操作は関数宣言、リゾルバーは定義と考えることができます。リゾルバーについては別のセクションで説明します。

この例は、スキーマがデータを操作する最も簡単な方法のみを示しています。GraphQL と AWS AppSyncの機能を活用して、複雑で堅牢でスケーラブルなアプリケーションを構築します。次のセクションでは、スキーマで使用できるさまざまなタイプとフィールドの動作をすべて定義します。

GraphQL タイプ

GraphQL は、さまざまなタイプをサポートします。前のセクションで見たように、タイプはデータの形状や動作を定義します。これらは GraphQL スキーマの基本的な構成要素です。

タイプは入力と出力に分類できます。入力は特殊なオブジェクトタイプ (Query、Mutation など) の引数として渡すことができるタイプですが、出力タイプはデータの保存と返しにのみ使用されます。タイプとその分類のリストは以下のとおりです。

- オブジェクト: オブジェクトにはエンティティを説明するフィールドが含まれます。例えば、book、authorName、publishingYear などの特性を記述するフィールドを持つオブジェクトのようなものが考えられます。これらは厳密には出力タイプです。

- スカラー: これらは int や string などのプリミティブ型です。通常はフィールドに割り当てられます。authorName フィールドを例にとると、「John Smith」のような名前を格納する String スカラーを割り当てることができます。スカラーは入力タイプでも出力タイプでもかまいません。
- 入力: 入力では、フィールドグループを引数として渡すことができます。オブジェクトとよく似た構造ですが、特別なオブジェクトに引数として渡すことができます。入力を使うと、スカラー、列挙型、その他の入力をスコープ内で定義できます。入力は入力タイプにしかなりません。
- 特殊オブジェクト: 特殊オブジェクトは状態を変更する操作を実行し、サービスの面倒な作業の大部分を行います。特殊なオブジェクトタイプには、クエリ、ミューテーション、サブスクリプションの 3 種類があります。通常、クエリはデータを取得し、ミューテーションはデータを操作し、サブスクリプションはクライアントとサーバー間の双方向接続を開いて維持し、常時通信を行います。特殊オブジェクトは、その機能上、入力でも出力でもありません。
- 列挙型: 列挙型はあらかじめ定義された有効な値のリストです。列挙型を呼び出す場合、その値はそのスコープで定義されている値のみになります。例えば、交通信号のリストを表すという trafficLights がある場合、それにはredLight や greenLight などの値が含まれることがありますが、purpleLight にはなり得ません。実際の信号機には信号の数が限られているため、列挙型を使用して信号を定義し、trafficLight 参照時にそれらだけが有効な値になるように強制できます。列挙型は入力型でも出力型でもかまいません。
- ユニオン/インターフェース: ユニオンを使うと、クライアントからリクエストされたデータに応じて 1 つ以上のものをリクエストで返すことができます。例えば、title フィールドのある Book 型と name フィールドのある Author 型がある場合、両方の型を結合することができます。クライアントが「ジュリアス・シーザー」というフレーズをデータベースから検索したい場合、ユニオンはジュリアス・シーザー (ウィリアム・シェイクスピアの戯曲) を Book title から、ジュリアス・シーザー(『コメンタリー・デ・ベロ・ガリコ』の作者) を Author name から返すことができます。ユニオンは出力タイプとしてのみ使用できます。

  インターフェースは、オブジェクトが実装しなければならないフィールドのセットです。これは、Java などのプログラミング言語のインターフェースに少し似ていますが、インターフェースで定義されたフィールドを実装する必要があります。例えば、Booktitleフィールドを含むというインターフェースを作成するとします。後で、Book を実装したもの Novel という型を作成したとします。Novel には title フィールドを含める必要があります。ただし、Novel には、pageCount または ISBN のような、インターフェイスにない他のフィールドも含めることができます。インターフェースは出力タイプにしかなれません。

以下のセクションでは、各タイプが GraphQL でどのように機能するかを説明します。

オブジェクト

GraphQL オブジェクトは、プロダクションコードでよく使われるタイプです。GraphQL では、オブジェクトは異なるフィールドの集まりであり (他の言語の変数と同様)、各フィールドは値を保持できる型 (通常はスカラーまたは別のオブジェクト) によって定義されます。オブジェクトは、サービス実装から取得/操作できるデータの単位です。

オブジェクトタイプは Type キーワードを使用して宣言されます。スキーマの例を少し変更してみましょう。

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}
```

ここでのオブジェクトタイプは Person と Occupation です。各オブジェクトには独自のフィールドと独自のタイプがあります。GraphQL の特徴の 1 つは、フィールドを他のタイプに設定できることです。Person の occupation フィールドには Occupation オブジェクトタイプが含まれていることがわかります。GraphQL はデータを記述するだけで、サービスの実装は記述していないため、この関連付けを行うことができます。

スカラー

スカラーは基本的に、値を保持するプリミティブ型です。には AWS AppSync、デフォルトの GraphQL スカラーとスカラーという 2 AWS AppSync 種類のスカラーがあります。スカラーは通常、オブジェクトタイプ内のフィールド値を格納するために使用されます。GraphQL のデフォルトタイプには Int、Float、String、Boolean、ID が含まれます。前の例をもう一度使ってみましょう。

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
```

```
}

type Occupation {
  title: String
}
```

name および title フィールドを除くと、どちらも String スカラーになります。Name は "John Smith" のような文字列値を返すことができ、タイトルは "firefighter" のようなものを返すことができます。一部の GraphQL 実装では、Scalar キーワードを使用してタイプの動作を実装するカスタムスカラーもサポートしています。ただし、 AWS AppSync は現時点ではカスタムスカラーをサポートしていません。スカラーのリストについては、「AWS AppSyncのスカラータイプ」を参照してください。

入力

入力型と出力型の概念により、引数を渡すときには一定の制限があります。一般的に渡す必要がある型、特にオブジェクトには制限があります。入力タイプを使用すると、このルールを回避できます。入力は、スカラー、列挙型、その他の入力タイプを含むタイプです。

入力は input キーワードを使用して定義されます。

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}

input personInput {
  id: ID!
  name: String
  age: Int
  occupation: occupationInput
}

input occupationInput {
  title: String
}
```

ご覧のとおり、元の型を模倣した入力を別々に設定できます。これらの入力は、次のようなフィールド操作でよく使用されます。

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}

input occupationInput {
  title: String
}

type Mutation {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person
}
```

まだ Occupation の代わりに occupationInput を渡して Person を作成することに注意してください。

これは入力の 1 つのシナリオに過ぎません。必ずしもオブジェクトを 1:1 でコピーする必要はありませんし、プロダクションコードでは、このような方法を使用することはほとんどありません。引数として入力する必要があるものだけを定義して、GraphQL スキーマを活用するとよいでしょう。

また、同じ入力を複数の操作で使用することもできますが、これはお勧めしません。スキーマの要件が変更された場合に備えて、各操作には入力の固有のコピーが含まれているのが理想的です。

特殊なオブジェクト

GraphQL は、スキーマがデータを取得/操作する方法に関するビジネスロジックの一部を定義する特別なオブジェクト用にいくつかのキーワードを予約しています。スキーマには、これらのキーワードがそれぞれ 1 つしか存在できません。これらは、クライアントが GraphQL サービスに対して実行するすべての要求されたデータのエントリポイントとして機能します。

特別なオブジェクトも type キーワードを使って定義されます。通常のオブジェクトタイプとは使い方が異なりますが、実装は非常に似ています。

Queries

クエリは、読み取り専用のフェッチを実行してソースからデータを取得するという点で GET オペレーションとよく似ています。GraphQL では、Query がサーバーに対してリクエストを行うクライアントのすべてのエントリポイントを定義します。GraphQL の実装には Query が必ずあります。

前のスキーマの例で使用した Query と変更されたオブジェクトタイプは次のとおりです。

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}
type Occupation {
  title: String
}
type Query {
  people: [Person]
}
```

Query には、データソースか Person インスタンスのリストを返す people というフィールドがあります。アプリケーションの動作を変更する必要があり、今度は別の目的で Occupation インスタンスのみのリストを返す必要があるとしましょう。これをクエリに追加するだけで済みます。

```
type Query {
  people: [Person]
  occupations: [Occupation]
}
```

GraphQL では、クエリをリクエストの単一ソースとして扱うことができます。お分かりのように、これは、異なるエンドポイントを使用して同じこと (.../api/1/people と .../api/1/occupations) を実現する RESTful な実装よりもずっと簡単になる可能性があります。

このクエリ用のリゾルバー実装があると仮定すると、実際のクエリを実行できるようになります。Query タイプは存在しますが、アプリケーションのコードで実行するには明示的に呼び出す必要があります。これは query キーワードを使用して行うことができます。

```
query getItems {
```

```
   people {
      name
   }
   occupations {
      title
   }
}
```

ご覧のとおり、このクエリは getItems と呼ばれ、people (Person オブジェクトのリスト) とoccupations (Occupation オブジェクトのリスト) を返します。people では、それぞれの Person の name フィールドのみを返し、それぞれの Occupation の title フィールドを返しています。また、レスポンスは次のようになります。

```
{
  "data": {
    "people": [
      {
        "name": "John Smith"
      },
      {
        "name": "Andrew Miller"
      },
      .
      .
      .
    ],
    "occupations": [
      {
        "title": "Firefighter"
      },
      {
        "title": "Bookkeeper"
      },
      .
      .
      .
    ]
  }
}
```

レスポンス例は、データがクエリの形状に従っていることを示している。取得された各エントリは、フィールドの範囲内で一覧表示されます。people と occupationsをそれぞれを個別のリ

ストとして返しています。便利ですが、ユーザーの名前と職業のリストを返すようにクエリを変更したほうが便利かもしれません。

```
query getItems {
   people {
      name
      occupation {
        title
      }
}
```

Person タイプには Occupation タイプの occupation フィールドが含まれているので、これは合法的な変更です。people の範囲内にリストされている場合は、title によって関連する Occupation とともにそれぞれの Person の name を返すことになります。また、レスポンスは次のようになります。

```
}
  "data": {
    "people": [
      {
        "name": "John Smith",
        "occupation": {
          "title": "Firefighter"
        }
      },
      {
        "name": "Andrew Miller",
        "occupation": {
          "title": "Bookkeeper"
        }
      },
      .
      .
      .
    ]
  }
}
```

Mutations

ミューテーションは、PUT や POST のような状態を変える操作に似ています。書き込み操作を実行してソース内のデータを変更し、レスポンスを取得します。データ変更リクエストのエントリ

ポイントを定義します。クエリとは異なり、ミューテーションはプロジェクトのニーズに応じてスキーマに含まれる場合と含まれない場合があります。スキーマの例からのミューテーションは次のとおりです。

```
type Mutation {
  addPerson(id: ID!, name: String, age: Int): Person
}
```

addPerson フィールドは、Person をデータソースに追加する 1 つのエントリポイントを表します。addPerson はフィールド名、id、name、age はパラメータ、Person は戻り型です。Person タイプを振り返ってみます。

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}
```

occupation フィールドを追加しました。ただし、オブジェクトは引数として渡すことができないため、このフィールドを直接 Occupation に設定することはできません。オブジェクトは厳密には出力タイプです。代わりに、同じフィールドを含む入力を引数として渡す必要があります。

```
input occupationInput {
  title: String
}
```

新しいインスタンスを作るときに、パラメータとして含めるように簡単に addPerson を更新することもできます。

```
type Mutation {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person
}
```

更新されたスキーマは次のとおりです。

```
type Person {
  id: ID!
  name: String
```

```
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}

input occupationInput {
  title: String
}

type Mutation {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person
}
```

元のオブジェクトの代わりに、occupation が occupationInput から title フィールドに渡すことで Person の作成が完了することに注意してください。addPerson のリゾルバーの実装があると仮定すると、実際のミューテーションを実行できるようになりました。Mutation 型は存在しますが、アプリケーションのコードで実行するには明示的に呼び出す必要があります。これは mutation キーワードを使用して行うことができます。

```
mutation createPerson {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput) {
    name
    age
    occupation {
      title
    }
  }
}
```

このミューテーションは createPerson と呼ばれ、addPerson がオペレーションです。新しい Person を作成するには id、name、age、occupation の引数を入力します。addPerson の範囲には、name、age などの他のフィールドもあります。これはあなたのレスポンスです。これらは addPerson オペレーションが完了した後に返されるフィールドです。この例の最後の部分は次のとおりです。

```
mutation createPerson {
  addPerson(id: "1", name: "Steve Powers", age: "50", occupation: "Miner") {
    id
```

```
    name
    age
    occupation {
      title
    }
  }
}
```

このミューテーションを使用すると、結果は以下のようになります。

```
{
  "data": {
    "addPerson": {
      "id": "1",
      "name": "Steve Powers",
      "age": "50",
      "occupation": {
        "title": "Miner"
      }
    }
  }
}
```

ご覧のとおり、レスポンスはリクエストした値を、ミューテーションで定義したのと同じ形式で返しました。混乱を減らし、今後さらにクエリが必要にならないように、変更されたすべての値を返すことをお勧めします。ミューテーションを使うと、複数の操作をその範囲に含めることができます。これらはミューテーションにリストされている順序で順番に実行されます。例えば、データソースに役職を追加する addOccupation という操作をもう 1 つ作成した場合、これを addPerson の後にミューテーションで呼び出すことができます。addPerson が最初に処理され、その後に addOccupation が処理されます。

Subscriptions

サブスクリプションは WebSockets を使用して、サーバーとクライアント間の永続的な双方向接続を開きます。通常、クライアントはサーバーをサブスクライブまたはリッスンします。サーバーがサーバー側で変更を加えたり、イベントを実行したりするたびに、サブスクライブしているクライアントは更新を受け取ります。このタイプのプロトコルは、複数のクライアントがサブスクライブしていて、サーバーや他のクライアントで発生した変更について通知を受ける必要がある場合に役立ちます。例えば、サブスクリプションを使用してソーシャルメディアフィードを更新できます。ユーザー A とユーザー B の 2 人のユーザーが、どちらもダイレクトメッセージを受信するたびに自動通知更新をサブスクライブしている場合があります。クライアント A

のユーザー A は、クライアント B のユーザー B にダイレクトメッセージを送信できます。ユーザー A のクライアントはダイレクトメッセージを送信し、サーバーによって処理されます。その後、サーバーはユーザー B のアカウントにダイレクトメッセージを送信し、クライアント B には自動通知を送信します。

スキーマの例に追加できる Subscription の例を以下に示します。

```
type Subscription {
  personAdded: Person
}
```

personAdded フィールドは、データソースに新しい Person が追加されるたびに、サブスクライブしているクライアントにメッセージを送信します。personAdded のリゾルバーの実装があると仮定すると、サブスクリプションを使用できるようになりました。Subscription 型は存在しますが、アプリケーションのコード内で実行するには明示的に呼び出す必要があります。これは subscription キーワードを使用して行うことができます。

```
subscription personAddedOperation {
  personAdded {
    id
    name
  }
}
```

サブスクリプションは personAddedOperation と呼ばれ、オペレーションは personAdded です。personAdded は新しい Person インスタンスの id および name フィールドを返します。ミューテーションの例を見てみると、以下の操作を使用して Person を追加しました。

```
addPerson(id: "1", name: "Steve Powers", age: "50", occupation: "Miner")
```

クライアントが新しく追加された Person へのアップデートを購読していた場合、addPerson 実行後に以下のように表示されるかもしれません。

```
{
  "data": {
    "personAdded": {
      "id": "1",
      "name": "Steve Powers"
    }
  }
```

```
}
```

以下は、サブスクリプションが提供するものの概要です。

サブスクリプションは、クライアントとサーバーが迅速で安定したアップデートを受信できるようにする双方向のチャネルです。通常、標準化された安全な接続を実現する WebSocket プロトコルを使用します。

サブスクリプションは、接続設定のオーバーヘッドを減らすという点で機敏です。一度サブスクライブすると、クライアントはそのサブスクリプションで長期間稼働し続けることができます。通常、開発者がサブスクリプションの有効期間を調整したり、要求される情報を設定したりできるようにすることで、コンピューティングリソースを効率的に使用します。

一般に、サブスクリプションを使用すると、クライアントは一度に複数のサブスクリプションを作成できます。サブスクリプションは AWS AppSync、 AWS AppSync サービスからリアルタイムの更新を受け取るためにのみ使用されます。クエリやミューテーションの実行には使用できません。

サブスクリプションに代わる主な方法はポーリングです。ポーリングでは、設定した間隔でクエリを送信してデータを要求します。このプロセスは通常、サブスクリプションほど効率的ではなく、クライアントとバックエンドの両方に大きな負担をかけます。

スキーマの例では言及されていなかったことの 1 つは、特殊なオブジェクト型も schema ルートで定義しなければならないという事実です。したがって、スキーマを に エクスポートすると AWS AppSync、次のようになります。

schema.graphql

```
schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}

.
.
.

type Query {
  # code goes here
}
```

```
type Mutation {
  # code goes here
}
type Subscription {
  # code goes here
}
```

列挙型

列挙型は、タイプやフィールドが持つ可能性のある有効な引数を制限する特殊なスカラーです。つまり、スキーマで列挙型が定義されると、それに関連するタイプまたはフィールドは列挙型内の値に限定されます。列挙型は文字列スカラーとしてシリアル化されます。プログラミング言語が異なれば、GraphQL 列挙型の処理も異なる場合があることに注意してください。例えば、JavaScript はネイティブの列挙型をサポートしていないため、代わりに列挙値を int 値にマップできます。

列挙型は `enum` キーワードを使用して定義されます。例を示します。

```
enum trafficSignals {
  solidRed
  solidYellow
  solidGreen
  greenArrowLeft
  ...
}
```

`trafficLights` 列挙型を呼び出すとき、引数に指定できるのは `solidRed`、`solidYellow`、`solidGreen` などのみです。列挙型を使うのは、はっきりしているが選択肢の数が限られているものを表すのに使うのが一般的です。

ユニオン/インターフェース

GraphQL の「Interface と Union」を参照してください。

GraphQL フィールド

フィールドはタイプのスコープ内に存在し、GraphQL サービスから要求された値を保持します。これらは、他のプログラミング言語の変数とよく似ています。例えば、次のような `Person` オブジェクトタイプがあります。

```
type Person {
   name: String
```

```
    age: Int
}
```

この場合のフィールドはそれぞれ name age で、String および Int 値が格納されます。上に示したようなオブジェクトフィールドは、クエリやミューテーションのフィールド (オペレーション) の入力として使用できます。例については、以下の Query を参照してください

```
type Query {
  people: [Person]
}
```

people フィールドは、データソースから Person のすべてのインスタンスを要求しています。GraphQL サーバーで Person を追加または取得すると、データはタイプとフィールドの形式に従うことが期待できます。つまり、スキーマ内のデータの構造によって、レスポンスでどのように構造化されるかが決まります。

```
}
  "data": {
    "people": [
      {
        "name": "John Smith",
        "age": "50"
      },
      {
        "name": "Andrew Miller",
        "age": "60"
      },
      .
      .
      .
    ]
  }
}
```

フィールドはデータを構造化する上で重要な役割を果たします。以下で説明するように、フィールドに適用してさらにカスタマイズできるプロパティがいくつかあります。

Lists

リストは指定されたタイプのすべての項目を返します。リストは括弧 [] を使ってフィールドの型に追加できます。

```
type Person {
  name: String
  age: Int
}
type Query {
  people: [Person]
}
```

Query では、Person を囲んでいる括弧は、データソースからの Person のすべてのインスタンスを配列として返したいことを示しています。レスポンスでは、それぞれの Personの name および age の値が 1 つの区切りリストとして返されます。

```
}
  "data": {
    "people": [
      {
        "name": "John Smith",         # Data of Person 1
        "age": "50"
      },
      {
        "name": "Andrew Miller",      # Data of Person 2
        "age": "60"
      },
      .                               # Data of Person N
      .
      .
    ]
  }
}
```

使用できるのは特殊なオブジェクトタイプだけではありません。リストは通常のオブジェクトタイプのフィールドでも使用できます。

NULL 以外

NULL 以外は、レスポンスで NULL であってはならないフィールドを示します。！記号を使用して、フィールドを NULL 以外に設定できます。

```
type Person {
  name: String!
  age: Int
}
```

```
type Query {
  people: [Person]
}
```

name フィールドを明示的に NULL にすることはできません。データソースにクエリを実行して、このフィールドに NULL を入力すると、エラーが発生します。

リストと NULL 以外を組み合わせることができます。次のクエリを比較してください。

```
type Query {
  people: [Person!]      # Use case 1
}

.
.
.

type Query {
  people: [Person]!      # Use case 2
}

.
.
.

type Query {
  people: [Person!]!     # Use case 3
}
```

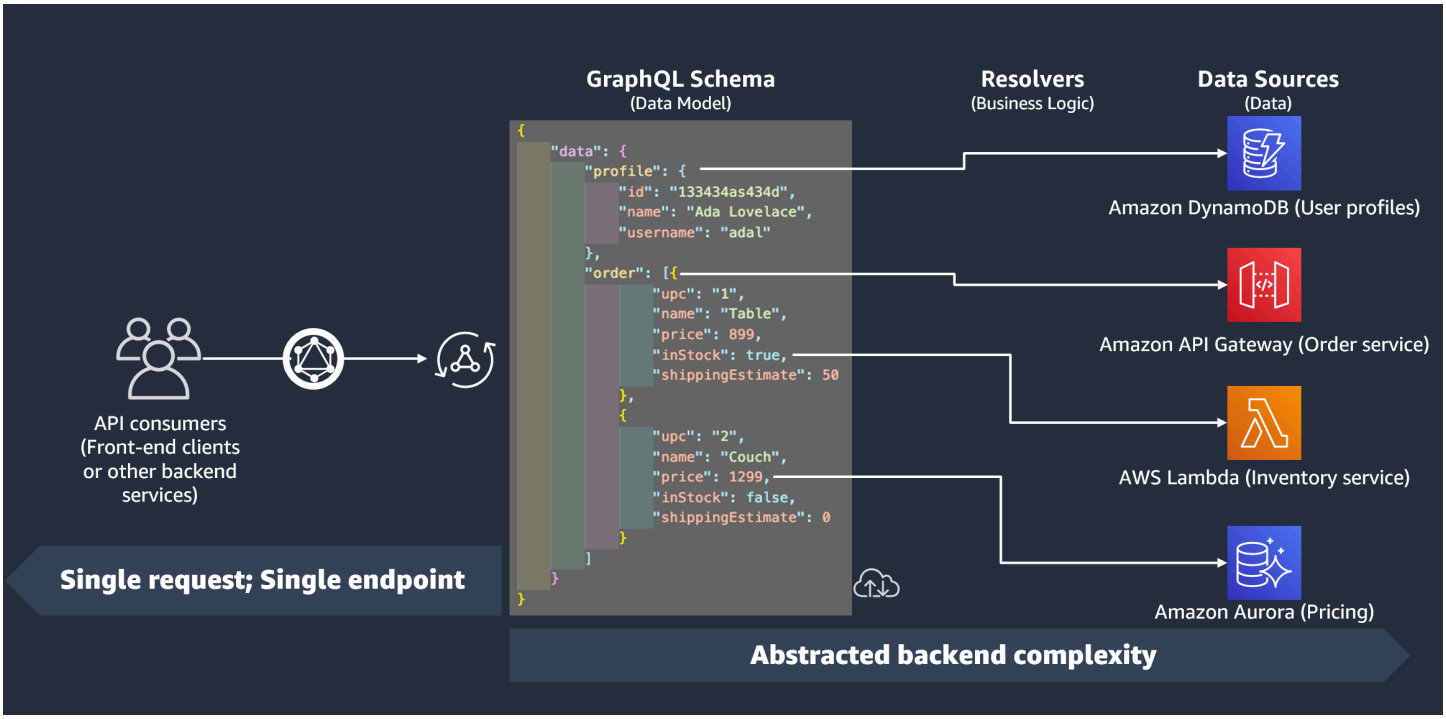
ユースケース 1 では、リストに NULL 項目を含めることはできません。ユースケース 2 では、リスト自体を NULL に設定することはできません。ユースケース 3 では、リストとその項目を NULL にすることはできません。ただし、いずれにしても、空のリストを返すことはできます。

ご覧のとおり、GraphQL には動くコンポーネントがたくさんあります。このセクションでは、単純なスキーマの構造と、スキーマがサポートするさまざまなタイプとフィールドを示しました。次のセクションでは、GraphQL API の他のコンポーネントと、それらがスキーマとどのように連携するかを説明します。

データソース

前のセクションでは、スキーマがデータの形状を定義することを学びました。ただし、そのデータがどこから来たのかについては説明していません。実際のプロジェクトでは、スキーマはサーバーへの

すべてのリクエストを処理するゲートウェイのようなものです。リクエストが行われると、スキーマはクライアントと接続する単一のエンドポイントとして機能します。スキーマはデータソースのデータにアクセスして処理し、クライアントに中継します。以下のインフォグラフィックを参照してください。



AWS AppSync と GraphQL は、バックエンドのフロントエンド (BFF) ソリューションを完全に実装します。これらは連携して、バックエンドを抽象化することで大規模な複雑さを軽減します。サービスで異なるデータソースやマイクロサービスを使用している場合は、各ソース (サブグラフ) のデータの形状を単一のスキーマ (スーパーグラフ) で定義することで、複雑さをある程度抽象化できます。つまり、GraphQL API は 1 つのデータソースの使用に限定されないということです。GraphQL API には任意の数のデータソースを関連付けて、それらがサービスとどのように相互作用するかをコードで指定できます。

インフォグラフィックでわかるように、GraphQL スキーマには、クライアントがデータを要求するために必要なすべての情報が含まれています。つまり、REST のように複数のリクエストを処理するのではなく、1 つのリクエストですべてを処理できるということです。これらのリクエストは、サービスの唯一のエンドポイントであるスキーマを通過します。リクエストが処理されると、リゾルバー (次のセクションで説明) がコードを実行して、関連するデータソースからのデータを処理します。レスポンスが返されると、データソースに関連付けられているサブグラフにスキーマ内のデータが入力されます。

AWS AppSync は、さまざまなデータソースタイプをサポートしています。以下の表では、各タイプについて説明し、それぞれの利点を列挙し、さらに詳しい情報を得るために役立つリンクを示します。

データソース	説明	利点	補足情報

Amazon DynamoDB	Amazon DynamoDB は、フルマネージド NoSQL データベースサービスであり、シームレスなスケーラビリティを備えた高速で予測可能なパフォーマンスを提供します。DynamoDB を使用すると、ディストリビューションデータベースの運用とスケーリングに伴う管理作業をまかせることができるため、ハードウェアのプロビジョニング、設定と構成、レプリケーション、ソフトウェアパッチ適用、クラスタースケーリングなどを自分で行う必要はなくなります。また、DynamoDB も保管時の暗号化を提供し、機密データの保護における負担と複雑な作業を解消します。	• 大規模環境でのパフォーマンス: DynamoDB は、どのような規模でも一貫したパフォーマンスを実現するように設計されています。これはパーティションを使用することで実現できます。DynamoDB はテーブルを複数の割り当てに自動的に分割し、複数のノードにまたがる複数の SSD に格納されます。これにより、一般的にネットワークスループットが向上し、レイテンシーが減少します。
		• 大規模なキャパシティ: DynamoDB はトラフィックを監視し、ネットワークが長時間にわたって過負荷状態になった場合にス
			• DynamoDB 公式ドキュメント
			  • パーティション
			  • Auto scaling
			  • 耐障害性
			  • モニタリング
			  • セキュリティ
			• GraphQL と DynamoDB
			• DynamoDB のリゾルバーオペレーション
			• 料金モデル

データソース	説明	利点	補足情報
		ループットを自動的にスケーリングできます。 • 可用性と耐障害性: DynamoDB は物理的に隔離された複数のリージョンでサポートされており、各リージョンには物理的に分離された複数のアベイラビリティーゾーンがあります。DynamoDB は、サービスが中断されると自動的にバックアップゾーンに切り替わります。データを保証するために、データを手動でバックアップおよび複製することもできます。 • ロギングとモニタリング: DynamoDB には、テーブル用の分析ツールがいくつか用意されています。テーブルのパフォーマンスをモニタリングし、サービスの大幅な変更を通知す	

データソース	説明	利点	補足情報
		るアラームを作成できます。 • セキュリティ: DynamoDB は厳格なプロトコルに従い、データが組織のセキュリティ要件に準拠していることを確認します。 • AWS AppSync との統合: DynamoDB は サービスとシームレスに統合されています。新しい DynamoDB テーブルを作成し、そこからスキーマを自動的に生成して、開発プロセスを効率化できます。また、リゾルバーのアカウント内の既存の DynamoDB テーブルからデータを簡単にリクエストできる一連のオペレーションも提供しています。	

データソース	説明	利点	補足情報

AWS Lambda	AWS Lambda は、サーバーのプロビジョニングや管理を行わずにコードを実行できるようにするコンピューティングサービスです。
	
	Lambda は可用性の高いコンピューティングインフラストラクチャでコードを実行し、コンピューティングリソースに関するすべての管理を行います。これには、サーバーおよびオペレーティングシステムのメンテナンス、容量のプロビジョニングおよび自動スケーリング、さらにログ記録などが含まれます。Lambda で必要なことは、サポートするいずれかの言語ランタイムにコードを与えることだけです。」
		• 従量課金制モデル: Lambda では、リソースを使用した場合にのみ課金されます。また、使用するリソースの量をアプリケーションのニーズに合わせて調整することもできます。
		• 自動スケーリング: アプリケーションが特定のプロセスのために追加の計算能力を必要とする場合があります。Lambda では、アプリケーションのニーズに合わせてコンピューティングリソースを自動的にスケーリングできます。
		• デプロイ時間の短縮: デプロイパッケージを使用すると開発プロセスを効率化できます。パッケージを使用して、関数コードを Lambda サービスにアップロードします。その後、そのランタイム環
			• 公式ドキュメント
			  • スケーリング
			  • デプロイメント
			  • ランタイム
			• Lambda リゾルバーチュートリアル
			• 料金モデル

データソース	説明	利点	補足情報
		境を使用して関数をテストし、実行できます。 • 汎用性: Lambda はさまざまなユースケースで使用できます。Lambda をサードパーティのサービスや AWS サービスと同様にシームレスに統合できます。例としては、CI/CD パイプラインや大量メールサービスなどがあります。 • AWS AppSync との統合: リゾルバーで Lambda 関数を簡単に呼び出してリクエストを処理できます。当社のサービスは、Lambda 呼び出しを実行するための効率的なリクエスト操作を提供します。シングルコールとバッチコールの両方が可能です。	

データソース	説明	利点	補足情報
OpenSearch	「Amazon OpenSearch Service は、 AWS クラウドでの OpenSearch クラスターのデプロイ、運用、スケーリングを容易にするマネージドサービスです。Amazon OpenSearch Service は、OpenSearch および従来の Elasticsearch OSS (ソフトウェアのファイナルオープンソースバージョンである 7.10 まで) をサポートしています。クラスターを作成するときに、どの検索エンジンを使用するかのオプションがあります。 OpenSearch は口グ分析、リアルタイムのアプリケーションモニタリング、クリックストリーム分析などのユースケース向けの、完全なオープンソースの検索および分析エンジンです。詳細については、「OpenSearch ド	• スケーリング: OpenSearch サーバーレスを使用すると、サービス要件に合わせてサービスを簡単にスケーリングできます。 • データ取り込み: OpenSearch Ingestion を使用してデータをインポート、処理、分析できます。データ取り込みには多数のアプリケーションがあり、こちらを参照してください。 • セキュリティ: OpenSearch は、IAM、CloudTrail、VPCs、認証などの AWS セキュリティ設定を管理できます。 • 可用性: OpenSearch はサービス内のさまざまなリージョンやアベイラビリティーゾーンもサポートしています。	• 公式ドキュメント • サーバーレス • 料金モデル

データソース	説明	利点	補足情報
	キュメント」を参照してください。 Amazon OpenSearch Service は、OpenSearch クラスターのすべてのリソースをプロビジョニングして、OpenSearch クラスターを起動します。また、障害が発生した OpenSearch Service ノードを自動的に検出して置き換え、セルフマネージドインフラストラクチャに関連するオーバーヘッドを減らします。また、単一の API コールを使用するか、コンソールで数回クリックするだけで、クラスターを簡単にスケーリングできます。	• AWS AppSync との統合: In AWS AppSync では、GraphQL APIs を使用して、アカウント内の既存の OpenSearch Service ドメインからデータを保存および取得できます。	

データソース	説明	利点	補足情報
HTTP エンドポイント	HTTP エンドポイントをデータソースとして使用できます。AWS AppSync は、パラメータやペイロードなどの関連情報を含むリクエストをエンドポイントに送信できます。HTTP レスポンスはリゾルバーに公開され、リゾルバーは操作終了後に最終レスポンスを返します。	• Lambda のようなサービスとそれほど統合されていない単純なアプリケーションに役立ちます。	• リゾルバーリファレンス

データソース	説明	利点	補足情報

Amazon EventBridge	EventBridge は、イベントを使用してアプリケーションコンポーネント同士を接続するサーバーレスサービスです。これにより、スケーラブルなイベント駆動型アプリケーションを簡単に構築できます。これを使用して、自社開発のアプリケーション、 AWS サービス、サードパーティーソフトウェアなどのソースから組織全体のコンシューマーアプリケーションにイベントをルーティングします。EventBridge では、イベントの取り込み、フィルタリング、変換、配信をシンプルかつ一貫性のある方法で行うことができるため、新しいアプリケーションをすばやく構築できます。	• イベント駆動型アーキテクチャ: イベント駆動型アーキテクチャを活用できます。
		• スケジュール: EventBridge Scheduler を使用すると、cron 式を使用してタスクやルールを自動化したり、イベントパターンの代わりに時間間隔を設定したりできます。
		• パイプ: EventBridge パイプを使用すると、イベントバスをパイプに置き換えることができます。パイプには、イベントをターゲットに送信する前に、追加のフィルタリングイベントパターンとデータ変換によるエンリッチメントが含まれます。
		• AWS AppSync:AppSync との統合により、リゾルバーを使用してイベントバスにイベント
			• 公式ドキュメント
			  • パイプ
			  • スケジューラー
			• リゾルバーリファレンス
			• 料金モデル

データソース	説明	利点	補足情報
		を送信できます。 AWS AppSync	

データソース	説明	利点	補足情報

リレーショナルデータベース	Amazon Relational Database Service (Amazon RDS)」は、AWS クラウドでのリレーショナルデータベースの設定、運用、スケーリングを容易にするウェブサービスです。業界スタンダードのリレーショナルデータベース向けに、費用対効果に優れた工クステンションを備え、一般的なデータベース管理タスクを管理します。」	• 管理が簡単に: RDS は定期的にリソースのメンテナンスを行います。通常、メンテナンスには DB インスタンスの基盤となるオペレーティングシステム (OS) やデータベースエンジンのバージョンのアップデートが伴います。通常の状況では、更新を実行するタイミングを決めることができます (セキュリティパッチは例外です)。
		• 推奨事項: RDS のレコメンデーション機能では、インスタンス内の潜在的な問題を解決するための提案が自動的に表示されます。
		• 可用性: RDS は世界中のさまざまな地域で使用できます。データベースのニーズをさまざまなノードに簡単に分散して、顧
			• 公式ドキュメント
			  • 特徴
			  • メンテナンス
			  • レコメンデーション
			  • ストレージオプション
			  • 現在利用できるリージョン
			  • セキュリティ
			• 料金モデル

データソース	説明	利点	補足情報
		客により良いサービスを提供できます。 • カスタマイズ: RDS は大企業の要件を満たすように調整されています。RDS には、コンピューティング、クイックデプロイ、スケーラビリティ、ストレージに関するさまざまなオプションが用意されています。 • セキュリティ: RDS は複数のツールやサービスと統合されており、ユーザー、データベース、ネットワークのレベルでデータベースのセキュリティを維持しています。 • AWS AppSync との統合: 成熟したバックエンドソリューションをお探しの場合、 AWS AppSync を使用すると、インスタンスをデータソー	

データソース	説明	利点	補足情報
		スとして使用してデータを送信、処理、保存、および返すことができます。	
none データソース	データソースサービスを使用する予定がない場合は、none に設定できます。none データソースは、設定後も明示的にデータソースとして分類されますが、ストレージメディアではありません。とはいえ、データ操作やパススルーには依然として役立つ場合があります。	• データ変換などに役立つかもしれません。 • 何かをローカルで解決する場合に便利です。	• リゾルバーリファレンス

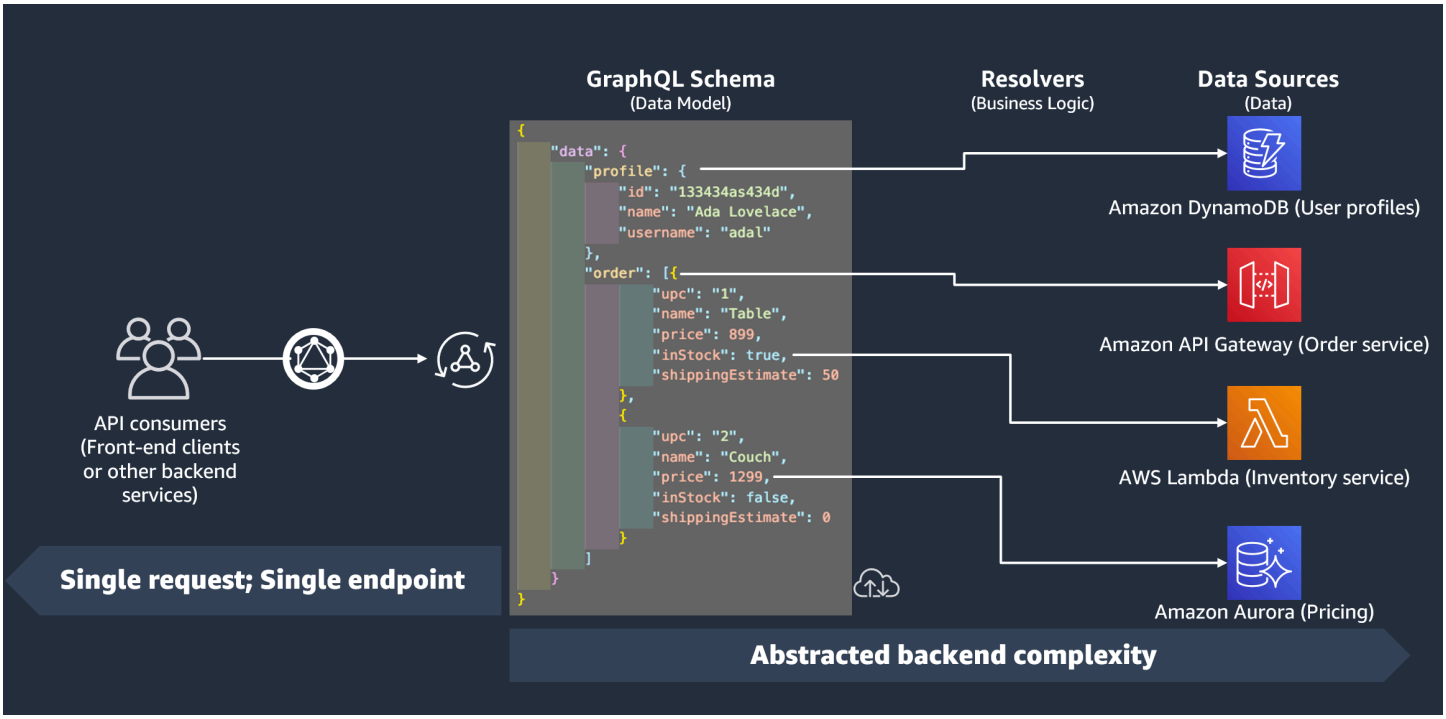
> **Tip**
>
> データソースとやり取りする方法の詳細については AWS AppSync、「データソースのアタッチ」を参照してください。

リゾルバー

前のセクションでは、スキーマとデータソースのコンポーネントについて学びました。次に、スキーマとデータソースがどのように相互作用するかを説明する必要があります。すべてはリゾルバーから始まります。

リゾルバーは、サービスにリクエストが送信されたときに、そのフィールドのデータをどのように解決するかを処理するコード単位です。リゾルバーは、スキーマのタイプ内の特定のフィールドにア

タッチされます。クエリ、ミューテーション、サブスクリプションフィールド操作の状態変更操作を実装するために最もよく使用されます。リゾルバーはクライアントのリクエストを処理し、結果を返します。結果はオブジェクトやスカラーのような出力タイプのグループでもかまいません。



リゾルバーランタイム

では AWS AppSync、まずリゾルバーのランタイムを指定する必要があります。リゾルバーランタイムは、リゾルバーが実行される環境を示します。また、リゾルバーが書き込まれる言語も指定します。 AWS AppSync 現在、 は APPSYNC_JS for JavaScript and Velocity Template Language (VTL) をサポートしています。JavaScript については「リゾルバーおよび関数の JavaScript runtime 機能」または VTL については「リゾルバーのマッピングテンプレートユーティリティーリファレンス」を参照してください。

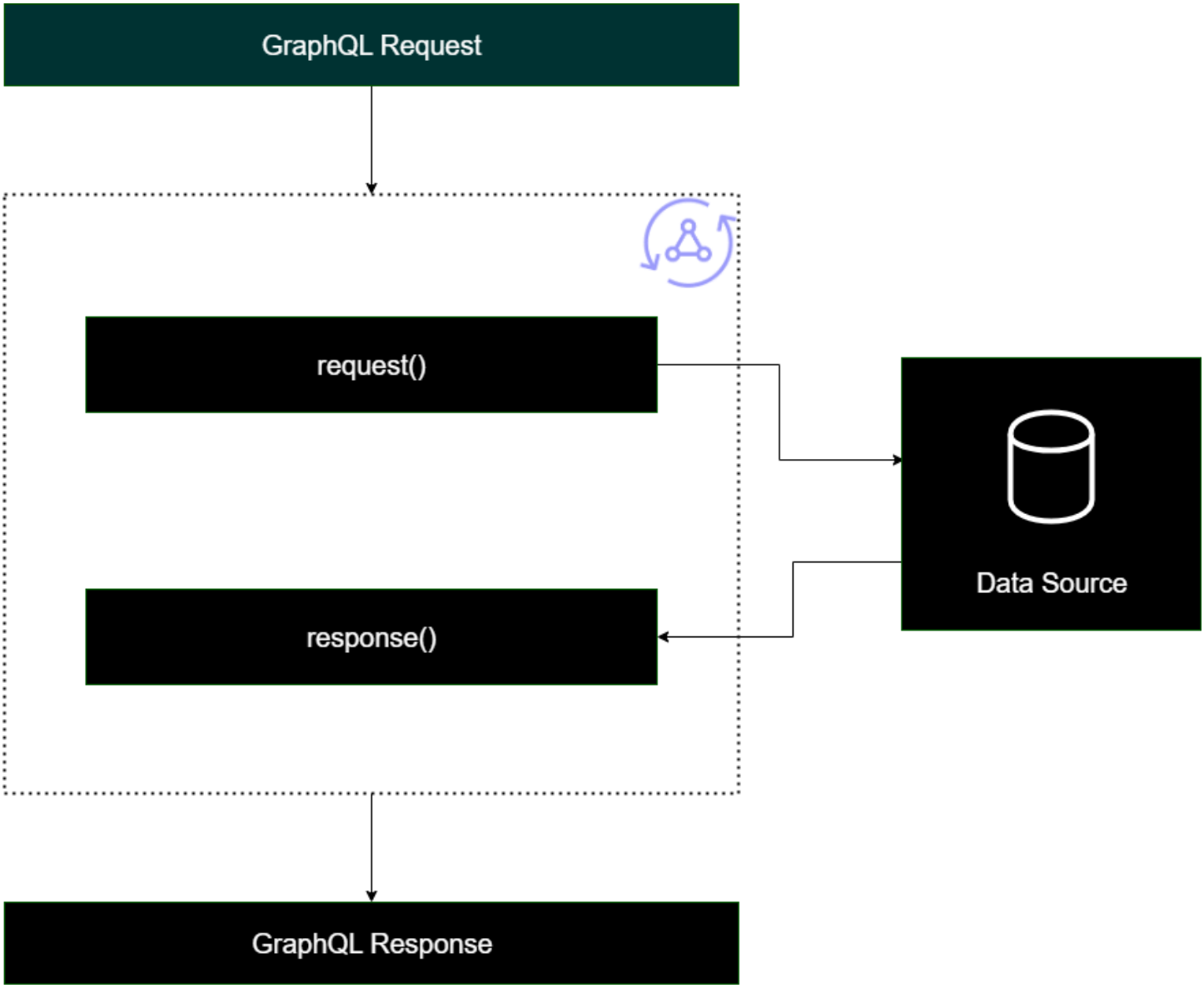
リゾルバーの構造

コード的には、リゾルバーはいくつかの方法で構造化できます。ユニット リゾルバーと パイプライン リゾルバーがあります。

ユニットリゾルバー

ユニットリゾルバーは、データソースに対して実行される単一のリクエストハンドラーとレスポンスハンドラーを定義するコードで構成されています。リクエストハンドラーはコンテキストオブジェクトを引数として受け取り、データソースの呼び出しに使用されたリクエストペイロードを返します。

レスポンスハンドラーは、実行されたリクエストの結果を含むペイロードをデータソースから受け取ります。レスポンスハンドラーは、ペイロードを GraphQL レスポンスに変換して GraphQL フィールドを解決します。

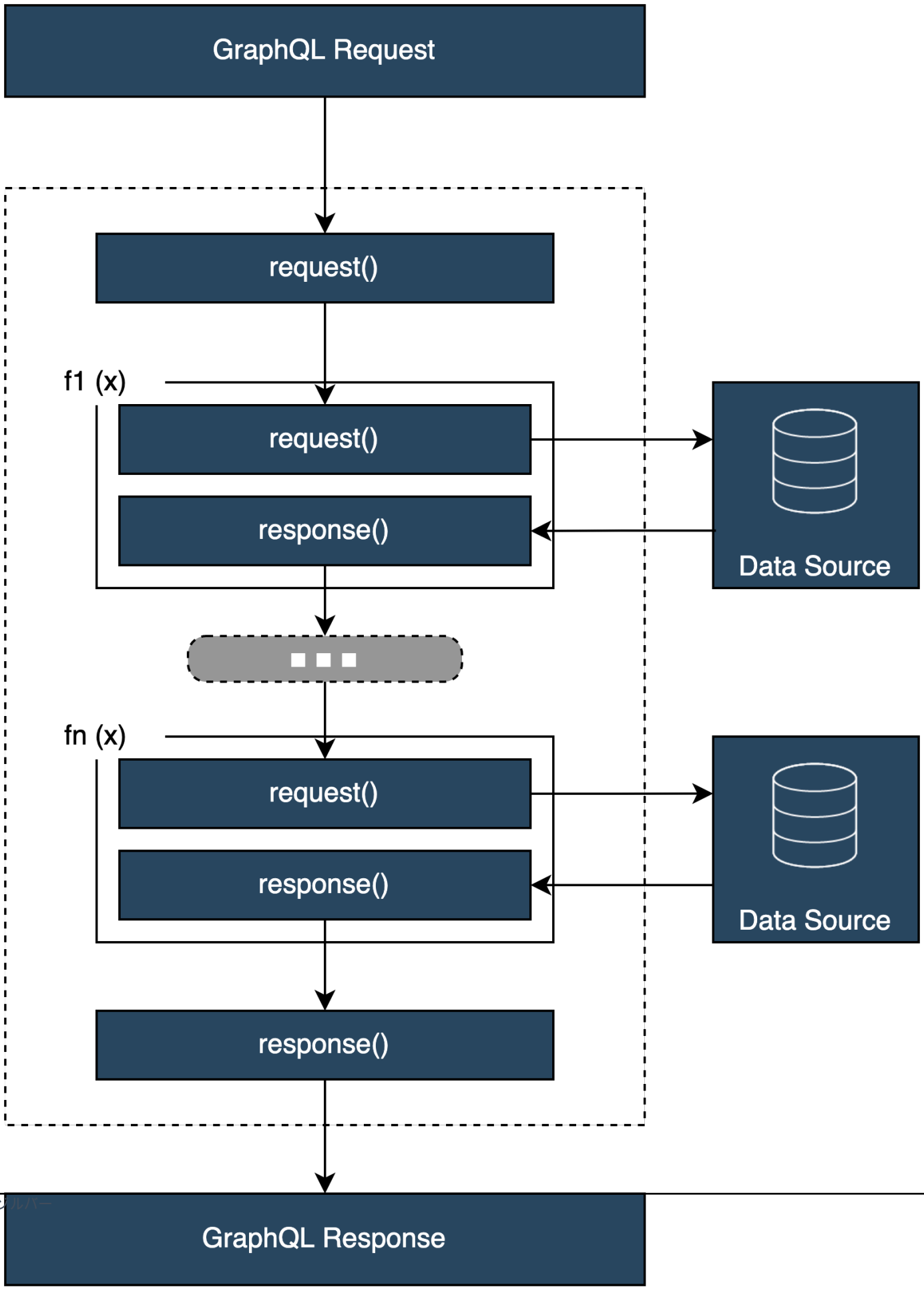


パイプラインリゾルバー

パイプラインリゾルバーを実装する場合、以下のような一般的な構造があります。

- before step: クライアントからリクエストが送信されると、使用されているスキーマフィールド (通常はクエリ、ミューテーション、サブスクリプション) のリゾルバーにリクエストデータが渡されます。リゾルバーは before step ハンドラーを使用してリクエストデータの処理を開始します。これにより、データがリゾルバーを通過する前に一部の前処理操作を実行できます。

- 関数: before ステップが実行されると、リクエストは関数リストに渡されます。リストの最初の関数がデータソースに対して実行されます。関数は、独自のリクエストハンドラーとレスポンスハンドラーを含むリゾルバーのコードのサブセットです。リクエストハンドラーは、リクエストデータを取得し、データソースに対して操作を実行します。レスポンスハンドラーは、データソースのレスポンスを処理してからリストに戻します。関数が複数ある場合、リクエストデータはリスト内の次に実行される関数に送信されます。リスト内の関数は、開発者が定義した順序で連続して実行されます。すべての関数が実行されると、最終結果は後のステップに渡されます。
- after step: after step は、GraphQL レスポンスに渡す前に、最終関数のレスポンスに対していくつかの最終オペレーションを実行できるハンドラー関数です。


GraphQL Request
request()
f1 (x)
request()
response()
Data Source
fn (x)
request()
response()
Data Source
response()
GraphQL Response

リゾルバーハンドラーの構造

ハンドラーは通常、Request および Response と呼ばれる関数です。

```
export function request(ctx) {
    // Code goes here
}

export function response(ctx) {
    // Code goes here
}
```

ユニットリゾルバーには、これらの関数のセットは 1 つしかありません。パイプラインリゾルバーには、処理前と処理後のステップ用にこれらのセットが 1 つあり、関数ごとに追加のセットがあります。これがどのようになるかを視覚化するために、単純な Query タイプを見てみましょう。

```
type Query {
 helloWorld: String!
}
```

これは String タイプの helloWorld というフィールドが 1 つある単純なクエリです。このフィールドには常に「Hello World」という文字列を返したいと仮定しましょう。この動作を実装するには、このフィールドにリゾルバーを追加する必要があります。ユニットリゾルバーには、次のようなものを追加できます。

```
export function request(ctx) {
    return {}
}

export function response(ctx) {
    return "Hello World"
}
```

データをリクエストしたり処理したりしていないので、request は空欄のままでも構いません。データソースは None だと仮定することもできます。つまり、このコードでは呼び出しを実行する必要がないということです。レスポンスは単に「Hello World」を返します。このリゾルバーをテストするには、次のクエリータイプを使用してリクエストを行う必要があります。

```
query helloWorldTest {
```

```
  helloWorld
}
```

これは helloWorld フィールドを返す helloWorldTest というクエリです。実行すると、helloWorld フィールドリゾルバーも実行され、レスポンスが返されます。

```
{
  "data": {
    "helloWorld": "Hello World"
  }
}
```

このような定数を返すのは一番簡単なことです。実際には、入力やリストなどを返すことになります。より複雑な例を次に示します。

```
type Book {
  id: ID!
  title: String
}

type Query {
  getBooks: [Book]
}
```

ここでは、Books のリストを返しています。本のデータの保存に DynamoDB テーブルを使用していると仮定しましょう。ハンドラーは以下のようになります。

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

このリクエストでは、組み込みのスキャン操作を使用してテーブル内のすべてのエントリを検索し、結果をコンテキストに保存して、レスポンスに渡しました。レスポンスは結果項目を受け取り、レスポンスとして返しました。

```
{
  "data": {
    "getBooks": {
      "items": [
        {
          "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
          "title": "book1"
        },
        {
          "id": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
          "title": "book2"
        },

        ...

      ]
    }
  }
}
```

リゾルバーのコンテキスト

リゾルバーでは、ハンドラーチェーンの各ステップが前のステップのデータの状態を認識している必要があります。あるハンドラーの結果を保存して、引数として別のハンドラーに渡すことができます。GraphQL は 4 つの基本的なリゾルバー引数を定義しています。

リゾルバーベース引数	説明
`obj` `root`、`parent`、など	親の結果です。
`args`	GraphQL クエリのフィールドに提供される引数。
`context`	すべてのリゾルバーに提供される値で、現在ログインしているユーザーやデータベースへのア

リゾルバーベース引数	説明
	クセスなどの重要なコンテキスト情報を保持します。
info	スキーマの詳細だけでなく、現在のクエリに関連するフィールド固有の情報を保持する値です。

では AWS AppSync、 context (ctx) 引数は上記のすべてのデータを保持できます。これはリクエストごとに作成されるオブジェクトで、認可情報、結果データ、エラー、リクエストメタデータなどのデータを含みます。コンテキストを使うと、プログラマーはリクエストの他の部分からのデータを簡単に操作できます。このスニペットをもう一度見てみましょう。

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

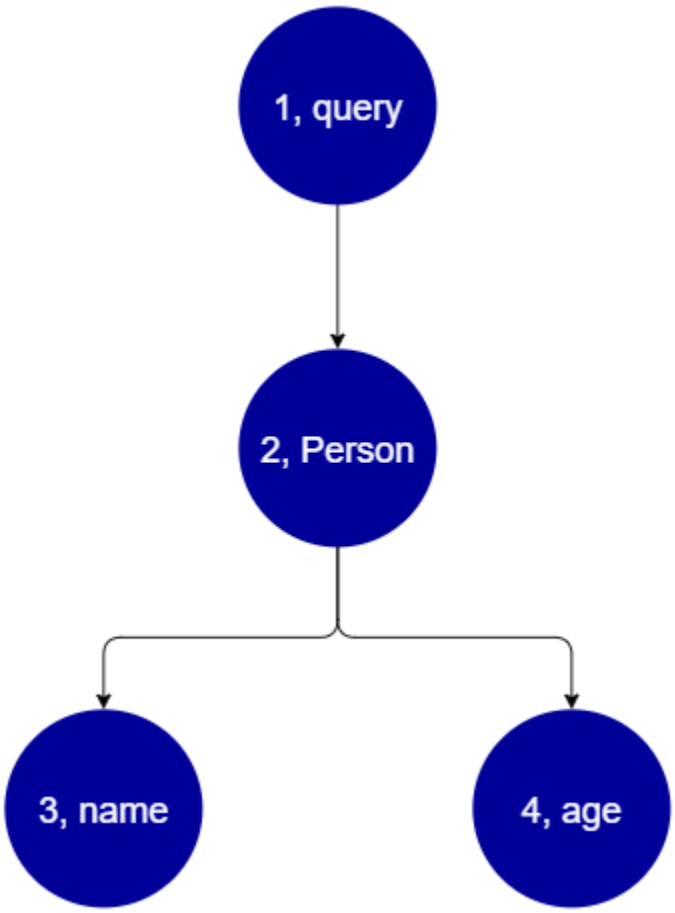
リクエストにはコンテキスト (ctx) が引数として渡されます。これがリクエストの状態です。テーブル内のすべての項目をスキャンし、その結果を resultのコンテキスト内の結果を格納します。次に、コンテキストが応答引数に渡され、応答引数が result にアクセスしてその内容を返します。

リクエストとパーシング

GraphQL サービスにクエリを行う場合、実行前に解析と検証のプロセスを実行する必要があります。リクエストは解析され、抽象構文ツリーに変換されます。ツリーの内容は、スキーマに対して複数の検証アルゴリズムを実行することによって検証されます。検証ステップの後、ツリーのノードがトラバースされ、処理されます。リゾルバーが呼び出され、結果がコンテキストに保存され、レスポンスが返されます。例えば、次のクエリを指定するとします。

```
query {
  Person {  //object type
    name  //scalar
    age   //scalar
  }
}
```

戻り値は name および age フィールドとともに Person を返します。このクエリを実行すると、ツリーは次のようになります。



ツリーから見ると、このリクエストはスキーマ内の Query をルートで検索しているようです。クエリの内部では、Person フィールドが解決されます。前の例から、これはユーザーからの入力であったり、値のリストなどが、必要なフィールド (name と age) を保持するオブジェクトタイプに関連付けられている可能性が高いことがわかっています。Personこの 2 つの子フィールドが見つかると、指定された順序 (name の後に age が続く) で解決されます。ツリーが完全に解決されると、リクエストは完了し、クライアントに返送されます。

GraphQL の追加プロパティ

GraphQL は、規模を問わずシンプルさと堅牢性を維持するためのいくつかの設計原則で構成されています。

宣言型

GraphQL は宣言型です。つまり、ユーザーはクエリしたいフィールドを宣言するだけでデータを記述 (整形) できます。レスポンスはこれらのプロパティのデータのみを返します。例えば、ISBN 13 id の値が *9780199536061* の DynamoDB テーブル内の Book オブジェクトを取得するオペレーションを次に示します。

```
{
  getBook(id: "9780199536061") {
    name
    year
    author
  }
}
```

このレスポンスでは、ペイロード内のフィールド (name、year、author) が返され、それ以外は何も返されません。

```
{
  "data": {
    "getBook": {
      "name": "Anna Karenina",
      "year": "1878",
      "author": "Leo Tolstoy",
    }
  }
}
```

この設計原則により、GraphQL は、複雑なシステムで REST API が処理するオーバーフェッチとアンダーフェッチという根強い問題を排除します。その結果、データ収集がより効率的になり、ネットワークパフォーマンスが向上します。

階層的

GraphQL は、要求されたデータをアプリケーションのニーズに合わせてユーザーが形作ることができるという点で柔軟です。リクエストされたデータは、常に GraphQL API で定義されたプロパティのタイプと構文に従います。例えば、次のスニペットは、Book *9780199536061* にリンクされているすべての保存済み引用文字列とページを返す quotes という新しいフィールドスコープを使った getBook オペレーションを示しています。

```
{
  getBook(id: "9780199536061") {
    name
    year
    author
    quotes {
      description
      page
    }
  }
}
```

このクエリを実行すると次の結果を返します。

```
{
  "data": {
    "getBook": {
      "name": "Anna Karenina",
      "year": "1878",
      "author": "Leo Tolstoy",
      "quotes": [
         {
            "description": "The highest Petersburg society is essentially one: in it
 everyone knows everyone else, everyone even visits everyone else.",
            "page": 135
         },
         {
            "description": "Happy families are all alike; every unhappy family is
 unhappy in its own way.",
            "page": 1
         },
         {
            "description": "To Konstantin, the peasant was simply the chief partner in
 their common labor.",
```

```
            "page": 251
          }
        ]
      }
    }
}
```

ご覧のように、リクエストされた本にリンクされている quotes フィールドは、クエリで記述されたのと同じ形式の配列として返されました。ここでは示していませんが、GraphQL には、取得するデータの場所にこだわらないという利点もあります。Books と quotes は別々に保存することもできますが、アソシエーションが存在する限り、GraphQL は引き続き情報を取得します。つまり、クエリは 1 回のリクエストで多数のスタンドアロンデータを取得できるということです。

イントロスペクティブ

GraphQL は自己文書化されている、つまりイントロスペクティブです。スキーマ内の基になる型やフィールドをユーザーが確認できるようにするいくつかの組み込み操作をサポートしています。例えば、Foo およびdate フィールドのある description 型があります。

```
type Foo {
 date: String
 description: String
}
```

_type オペレーションを使うと、スキーマの下にあるタイピングメタデータを検索できます。

```
{
  __type(name: "Foo") {
    name                    # returns the name of the type
    fields {                # returns all fields in the type
      name                  # returns the name of each field
      type {                # returns all types for each field
        name                # returns the scalar type
      }
    }
  }
}
```

これは応答を返します。

```
{
```

```
  "__type": {
    "name": "Foo",                    # The type name
    "fields": [
      {
        "name": "date",               # The date field
        "type": { "name": "String" }   # The date's type
      },
      {
        "name": "description",        # The description field
        "type": { "name": "String" }   # The description's type
      },
    ]
  }
}
```

この機能は、特定の GraphQL スキーマがどのタイプとフィールドをサポートしているかを調べるために使用できます。GraphQL は、さまざまなイントロスペクティブなオペレーションをサポートしています。詳細については、「イントロスペクション」を参照してください。

強力なタイピング

GraphQL は、型と項目のシステムを通じて厳密な型指定をサポートしています。スキーマで何かを定義する場合は、実行前に検証できる型でなければなりません。また、GraphQL の構文仕様に従わなければなりません。この概念は他の言語のプログラミングと何ら変わりはありません。例えば、先ほどの、Foo 型があります。

```
type Foo {
 date: String
 description: String
}
```

Foo が作成されるオブジェクトであることがわかります。Foo のインスタンス内には、date および description フィールドがあり、両方とも Stringプリミティブ型 (スカラー) です。構文的には、Foo が宣言されていて、そのフィールドがスコープ内に存在していることがわかります。このように型の確認と論理構文を組み合わせることで、GraphQL API は簡潔でわかりやすいものになります。GraphQL のタイピングと構文の仕様については、こちらを参照してください。

開始方法: AWS AppSync で最初の GraphQL API を作成する

AWS AppSync コンソールを使用して、GraphQL API を設定および起動できます。GraphQL API には通常、次の 3 つのコンポーネントが必要です。

1. GraphQL スキーマ - GraphQL スキーマは API の設計図です。操作の実行時にリクエストできるタイプとフィールドを定義します。スキーマにデータを入力するには、データソースを GraphQL API に接続する必要があります。このクイックスタートガイドでは、定義済みのモデルを使用してスキーマを作成します。
2. データソース - GraphQL API にデータを入力するためのデータを含むリソースです。これは、DynamoDB テーブル、Lambda 関数などです。 AWS AppSync は、堅牢でスケーラブルな GraphQL APIs を構築するための多数のデータソースをサポートしています。データソースはスキーマのフィールドにリンクされています。フィールドに対してリクエストが実行されるたびに、ソースからのデータがフィールドに入力されます。このメカニズムはリゾルバーによって制御されます。このクイックスタートガイドでは、スキーマと一緒に定義済みのモデルを使用してデータソースを作成します。
3. リゾルバー - リゾルバーはスキーマフィールドをデータソースにリンクする役割を担います。ソースからデータを取得し、 フィールドで定義された内容に基づいて結果を返します。 AWS AppSync は、GraphQL APIs のリゾルバーを記述するための JavaScript と VTL の両方をサポートしています。このクイックスタートガイドでは、リゾルバーはスキーマとデータソースに基づいて自動的に生成されます。このセクションではこれについては詳しく説明しません。

AWS AppSync は、すべての GraphQL コンポーネントの作成と設定をサポートしています。コンソールを開くと、以下の方法を使用して API を作成できます。

1. 事前定義されたモデルを使用してカスタマイズされた GraphQL API を生成し、それをサポートする新しい DynamoDB テーブル (データソース) を設定して設計します。
2. 空白のスキーマを使用し、データソースやリゾルバーなしで GraphQL API を設計します。
3. DynamoDB テーブルを使用してデータをインポートし、スキーマのタイプとフィールドを生成します。
4. AWS AppSync の WebSocket 機能と Pub/Sub アーキテクチャを使用して、リアルタイム APIs。
5. 既存の GraphQL API (ソース API) を使用して Merged API にリンクします。

> **Note**
>
> より高度なツールを使用する前に、「スキーマの設計」セクションを確認することをおすすめします。これらのガイドでは、 AWS AppSync でより複雑なアプリケーションを構築するために概念的に使用できるより簡単な例について説明します。

AWS AppSync は、GraphQL APIs。具体的には次のとおりです。

1. AWS Amplify
2. AWS SAM
3. AWS CloudFormation
4. CDK

次の例は、事前定義されたモデルと DynamoDB を使用して GraphQL API の基本的なコンポーネントを作成する方法を示しています。

トピック

- AWS AppSync コンソールでのスキーマの起動
- AWS AppSync コンソールのツアー
- GraphQL ミューテーションを使用して AWS AppSync コンソールの DynamoDB テーブルにデータを追加する
- GraphQL クエリを使用して AWS AppSync コンソールの DynamoDB テーブルからデータを取得する
- AWS AppSync コンソールの補足セクション

AWS AppSync コンソールでのスキーマの起動

この例では、ユーザーが「######」や「########」など、毎日の雑用リマインダー用の Todo アイテムを作成できる Todo API を作成します。このアプリは、状態が Amazon DynamoDB に保持されている GraphQL オペレーションの使用方法のデモを示します。

概念的には、初めての GraphQL API を作成するには 3 つの主要なステップがあります。スキーマ (タイプとフィールド) を定義し、データソースをフィールドにアタッチして、ビジネスロジックを処

理するリゾルバーを作成する必要があります。ただし、コンソールの操作では順序が変わります。まず、データソースとスキーマの相互作用を定義し、後でスキーマとリゾルバーを定義します。

GraphQL API を作成するには

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
2. ダッシュボードで、[API の作成] を選択します。
3. [GraphQL API] が選択されている状態で、[最初から設計] を選択します。[次へ] を選択します。
4. [API 名] では、あらかじめ入力されている名前を **Todo API** に変更し、[次へ] を選択します。

> **Note**
>
> 他にもオプションがありますが、この例では使用しません。

5. [ターゲットリソース] セクションで、以下の操作を行います。

   a. [DynamoDB テーブルを基にしたタイプを今すぐ作成] を選択します。

   > **Note**
   >
   > つまり、データソースとしてアタッチする新しい DynamoDB テーブルを作成します。

   b. [名前] フィールドに **Todo** を入力します。

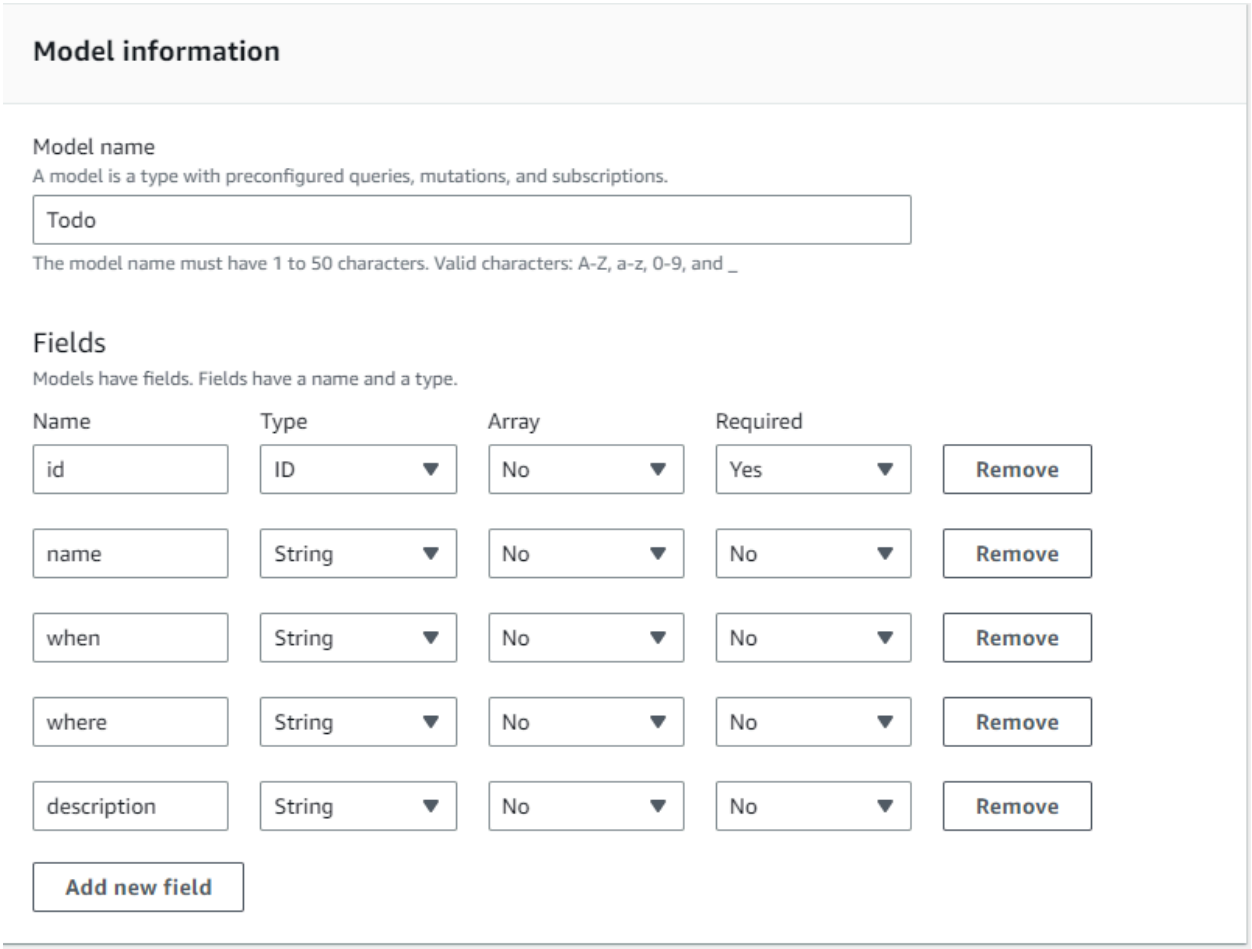
   > **Note**
   >
   > 最初の要件は、スキーマを定義することです。このモデル名は型名になるので、実際に行うのはスキーマに存在する Todo という type を作成することです。
   >
   > ```
   > type Todo {}
   > ```

   c. [フィールド] で、次の操作を行います。

      i. **id** という名前で、タイプは ID、必須は Yes に設定して設定します。

> **Note**
>
> これらは指定したタイプ Todo の範囲内に存在するフィールドです。ここでのフィールド名は、id でタイプは ID! です。
>
> ```
> type Todo {
>  id: ID!
> }
> ```
>
> AWS AppSync は、さまざまなユースケースで複数のスカラー値をサポートしています。

ii. [新しいフィールドを追加] を使用して、Name 値を **name**、**when**、**where**、**description** に設定した 4 つの追加フィールドを作成します。それらの Type 値は String になり、Array および Required の値は両方とも No に設定されます。次のようになります。



> **Note**
>
> フルタイプとそのフィールドは次のようになります。
>
> ```
> type Todo {
>  id: ID!
>  name: String
>  when: String
>  where: String
>  description: String
> }
> ```
>
> この定義済みモデルを使用してスキーマを作成しているため、データソースへの入力が容易になるように、create、delete update などの型に基づく定型的な変更もいくつか追加されます。

d. [モデルテーブルの設定] に、**TodoAPITable** などのテーブル名を入力します。プライマリキーを id に設定します。

> **Note**
>
> 基本的に、*TodoApitable* という新しい DynamoDB テーブルを作成します。このテーブルは、プライマリデータソースとして API にアタッチされます。プライマリキーは、この前に定義した必須 id フィールドに設定されます。この新しいテーブルは空白で、パーティションキー以外は何も含まれていないことに注意してください。

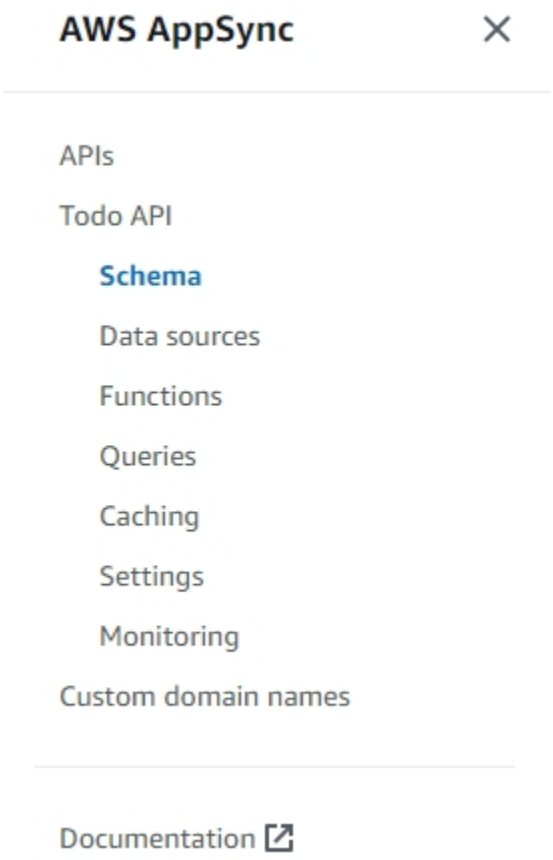
e. 次に、[次へ] を選択します。

6. 選択した内容を確認し、[ロールを作成] をクリックします。しばらく待ってから、 AWS AppSync サービスが API の作成を完了します。

スキーマと DynamoDB データソースを含む GraphQL API が正常に作成されました。上記の手順を要約すると、まったく新しい GraphQL API を作成することにしました。API の名前を定義し、次に最初のタイプを追加してスキーマ定義を追加しました。タイプとそのフィールドを定義し、データを含まない新しい DynamoDB テーブルを作成して、いずれかのフィールドにデータソースをアタッチすることにしました。

AWS AppSync コンソールのツアー

DynamoDB テーブルにデータを追加する前に、 AWS AppSync コンソールエクスペリエンスの基本機能を確認する必要があります。ページの左側にある AWS AppSync コンソールタブを使用すると、 AWS AppSync が提供する主要なコンポーネントや設定オプションに簡単に移動できます。



スキーマデザイナー

[スキーマ] を選択すると、作成したばかりのスキーマが表示されます。スキーマのコンテンツを確認すると、開発プロセスを効率化するための一連のヘルパー操作が既に読み込まれていることがわかります。スキーマエディターでコードをスクロールすると、前のセクションで定義したモデルにたどり着きます。

```
type Todo {
 id: ID!
 name: String
 when: String
 where: String
 description: String
```

```
}
```

作成したモデルが、スキーマ全体で使用されていた基本タイプになりました。まず、このタイプから自動的に生成されたミューテーションを使用して、データソースにデータを追加します。

スキーマエディターに関するその他のヒントや情報をいくつか紹介します。

1. コードエディタには、独自のアプリを記述するときに使用できるリンティングとエラーチェックの機能があります。
2. コンソールの右側には、作成された GraphQL 型と、クエリなどの別の最上位の型におけるリゾルバーが表示されます。
3. スキーマに新しい型を追加する場合 (例: `type User {...}`)、 AWS AppSync で DynamoDB リソースをプロビジョニングできます。これらには、適切なプライマリキー、ソートキー、およびインデックス設計が含まれ、GraphQL データアクセスパターンに最も一致します。上部にある [Create Resources (リソースの作成)] を選択し、メニューからいずれかのユーザー定義タイプを選択すると、スキーマ設計でさまざまなフィールドオプションを選択できます。これについては、「スキーマの設計」セクションで説明します。

リゾルバーの設定

スキーマデザイナーの [リゾルバー] セクションには、スキーマのすべてのタイプとフィールドが含まれます。フィールドのリストをスクロールすると、[アタッチ] を選択することで特定のフィールドにリゾルバーをアタッチできることがわかります。これにより、リゾルバーコードを記述できるコードエディタが開きます。 AWS AppSync は VTL ランタイムと JavaScript ランタイムの両方をサポートしています。このランタイムは、Actions を選択してページの上部で変更し、次にランタイムを更新します。ページの下部には、複数の操作を順番に実行する関数を作成することもできます。ただし、リゾルバーは上級者向けのトピックなので、このセクションでは説明しません。

データソース

[データソース] を選択して DynamoDB テーブルを表示します。`Resource` オプション (利用可能な場合) を選択すると、データソースの設定を表示できます。この例では、これは DynamoDB コンソールにつながります。そこから、データを編集できます。データソースを選択し、[編集] を選択することで、一部のデータを直接編集することもできます。データソースを削除する必要が生じた場合は、データソースを選択して [削除] を選択できます。最後に、[データソースの作成] を選択し、名前とタイプを設定することで、新しいデータソースを作成できます。このオプションは、 AWS AppSync サービスを既存のリソースにリンクするためのものです。 AWS AppSync が認識する前に、関連するサービスを使用してアカウントでリソースを作成する必要があります。

クエリ

[クエリ] を選択すると、クエリとミューテーションが表示されます。モデルを使用して GraphQL API を作成すると、 AWS AppSync はテスト目的でヘルパーミューテーションとクエリを自動的に生成します。クエリエディタの左側には Explorer があります。これはすべてのミューテーションとクエリを示すリストです。ここで使いたい操作やフィールドは、名前の値をクリックすることで簡単に有効にできます。これにより、エディタの中央にコードが自動的に表示されます。ここでは、値を変更してミューテーションやクエリを編集できます。エディターの下部には、操作の入力変数のフィールド値を入力できるクエリ変数エディターがあります。エディター上部の [実行] を選択すると、実行するクエリ/ミューテーションを選択するドロップダウンリストが表示されます。この実行の出力は、ページの右側に表示されます。上部の [エクスプローラー] セクションに戻り、操作 (クエリ、ミューテーション、サブスクリプション) を選択し、[+] 記号を選択してその特定の操作の新しいインスタンスを追加できます。ページの上部には、クエリ実行の認可モードを含む別のドロップダウンリストが表示されます。ただし、このセクションではその機能については説明しません (詳細については、「セキュリティ」を参照してください)。

設定

[設定] を選択すると、GraphQL API のいくつかの設定オプションが表示されます。ここでは、ロギング、トレース、ウェブアプリケーションファイアウォール機能などのオプションを有効にできます。また、新しい認可モードを追加して、データが外部に流出するのを防ぐこともできます。ただし、これらのオプションはより高度なものなので、このセクションでは説明しません。

> **Note**
>
> デフォルト認可モード、`API_KEY` では、API キーを使用して、アプリケーションをテストします。これは、新しく作成されたすべての GraphQL API に与えられる基本認可です。別の製造方法を使用することをお勧めします。このセクションの例では API キーのみを使用します。サポートされている認可方法の詳細については、「セキュリティ」を参照してください。

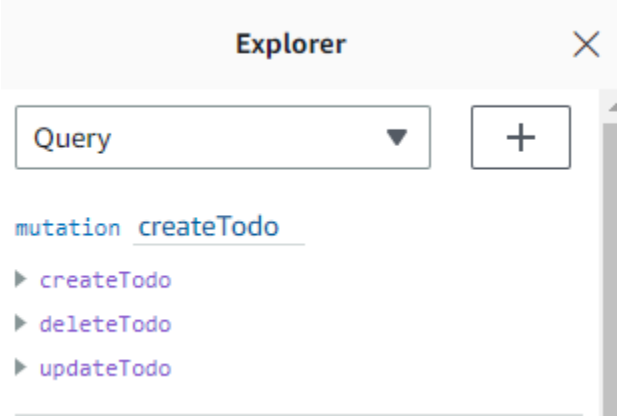
GraphQL ミューテーションを使用して AWS AppSync コンソールの DynamoDB テーブルにデータを追加する

次のステップは、GraphQL ミューテーションを使用して空の DynamoDB テーブルにデータを追加することです。ミューテーションは GraphQL の基本的な操作タイプの 1 つです。これらはスキー

マで定義され、データソース内のデータを操作できます。REST API に関しては、これらは PUT や POST などの操作と非常に似ています。

データソースにデータを追加するには

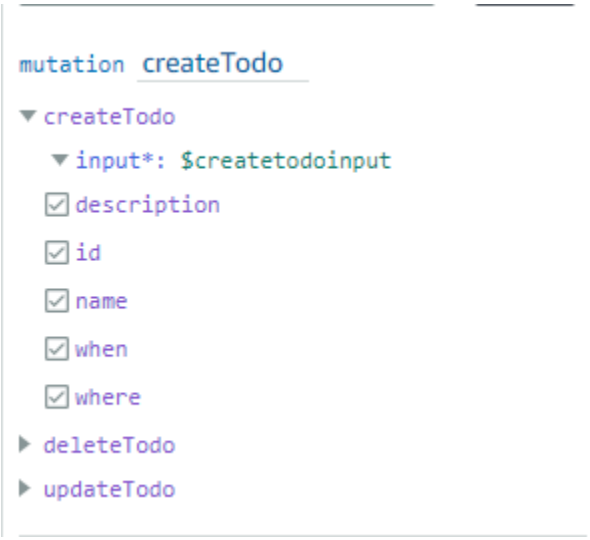
1. まだサインインしていない場合は、 にサインイン AWS Management Console して AppSync コンソールを開きます。
2. テーブルから API を選択します。
3. 左側のタブで [クエリ] を選択します。
4. 表の左側にある [エクスプローラー] タブには、クエリエディターですでに定義されているミューテーションやクエリがいくつか表示されている場合があります。



> **Note**
>
> このミューテーションは、実際には Mutation タイプとしてスキーマに存在しています。これには次のようなコードがあります。
>
> ```
> type Mutation {
>  createTodo(input: CreateTodoInput!): Todo
>  updateTodo(input: UpdateTodoInput!): Todo
>  deleteTodo(input: DeleteTodoInput!): Todo
> }
> ```
>
> ご覧のとおり、ここでの操作はクエリエディター内の操作と似ています。

AWS AppSync は、前に定義したモデルからこれらを自動的に生成しました。この例では、createTodo ミューテーションを使用して *TodoApitable* テーブルにエントリを追加します。

5. createTodo ミューテーションの下で createTodo 操作を展開して選択します。



上の図のように、すべてのフィールドのチェックボックスを有効にします。

> **Note**
>
> ここに表示される属性は、ミューテーションのさまざまな変更可能な要素です。input は、createTodo のパラメータと考えることができます。チェックボックス付きのさまざまなオプションは、操作が実行されるとレスポンスで返されるフィールドです。

6. 画面中央のコードエディターで、操作が createTodo ミューテーションの下に表示されているのがわかります。

```
mutation createTodo($createtodoinput: CreateTodoInput!) {
  createTodo(input: $createtodoinput) {
    where
    when
    name
    id
    description
  }
}
```

Note

このスニペットを正しく説明するには、スキーマコードも確認する必要があります。宣言 mutation createTodo($createtodoinput: CreateTodoInput!){} は、その操作の 1 つによるミューテーションです。createTodoすべてのミューテーションはスキーマにあります。

```
type Mutation {
 createTodo(input: CreateTodoInput!): Todo
 updateTodo(input: UpdateTodoInput!): Todo
 deleteTodo(input: DeleteTodoInput!): Todo
}
```

エディターからのミューテーション宣言に戻ると、パラメータは CreateTodoInput の必須の入力タイプで $createtodoinput というオブジェクトです。CreateTodoInput (およびミューテーション内のすべての入力) もスキーマで定義されていることに注意してください。例えば、CreateTodoInput のボイラープレートコードは以下のとおりです。

```
input CreateTodoInput {
 name: String
 when: String
 where: String
 description: String
}
```

これには、モデルで定義したフィールド、name、when、where および descriptionが含まれています。
エディタのコードに戻ると,createTodo(input: $createtodoinput) {} では入力を $createtodoinput として宣言します。これはミューテーション宣言でも使用されていました。これは、GraphQL が入力を指定された型と照合して検証し、正しい入力で使用されていることを確認できるためです。
エディターコードの最後の部分には、操作の実行後にレスポンスで返されるフィールドが表示されます。

```
{
    where
    when
```

```
    name
    id
    description
  }
```

このエディターの下の [クエリ変数] タブには、以下のデータを含む汎用 createtodoinput オブジェクトが表示されます。

```
{
  "createtodoinput": {
    "name": "Hello, world!",
    "when": "Hello, world!",
    "where": "Hello, world!",
    "description": "Hello, world!"
  }
}
```

> **Note**
>
> ここで、前述の入力の値を割り当てます。
>
> ```
> input CreateTodoInput {
>  name: String
>  when: String
>  where: String
>  description: String
> }
> ```

DynamoDB テーブルに入れたい情報を追加して createtodoinput を変更します。今回は、リマインダーとしていくつかの Todo 項目を作成したいと考えました。

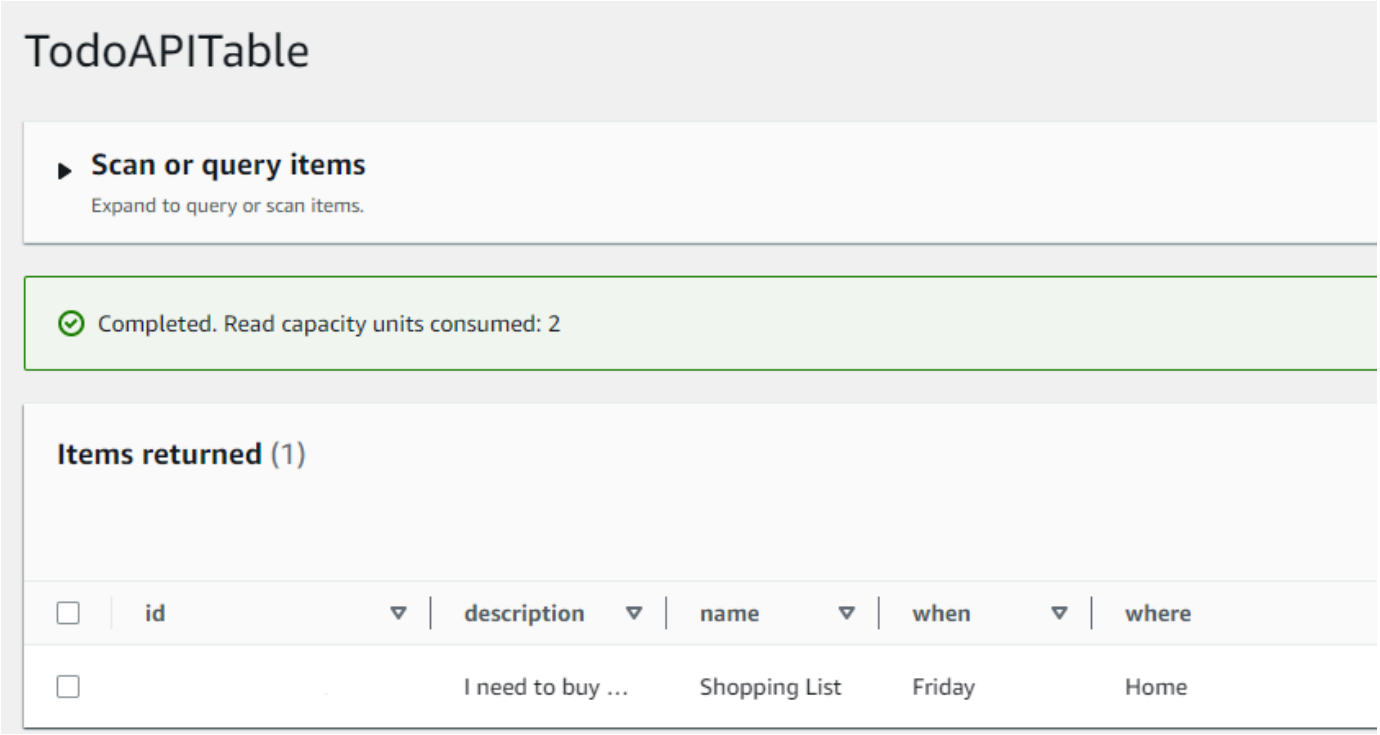
```
{
  "createtodoinput": {
    "name": "Shopping List",
    "when": "Friday",
    "where": "Home",
    "description": "I need to buy eggs"
```

```
  }
}
```

7. エディター上部の [実行] を選択します。ドロップダウンリストから [createTodo] を選択します。エディタの右側に、レスポンスが表示されます。次のように表示されます。

```
{
  "data": {
    "createTodo": {
      "where": "Home",
      "when": "Friday",
      "name": "Shopping List",
      "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
      "description": "I need to buy eggs"
    }
  }
}
```

DynamoDB サービスに移動すると、データソースに次の情報を含むエントリが表示されます。



操作を要約すると、GraphQL エンジンがレコードを解析し、リゾルバーがそのレコードを Amazon DynamoDB テーブルに挿入しました。これも DynamoDB コンソールで確認できます。id 値を渡す必要がないことに注意してください。id が生成され、結果に返されます。この例では、DynamoDB

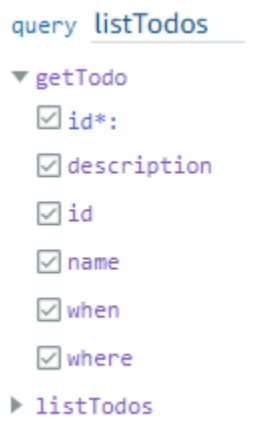
リソースに設定されているパーティションキーに対して、GraphQL リゾルバーの `autoId()` 関数を使用しているためです。リゾルバーを構築する方法については、別のセクションで説明します。戻り `id` 値を書き留めておきます。次のセクションでは、GraphQL クエリでデータを取得するときに使用します。

GraphQL クエリを使用して AWS AppSync コンソールの DynamoDB テーブルからデータを取得する

これで、データベースにレコードがあるので、クエリを実行すると結果が返されます。クエリは、GraphQL の他の基本的な操作の 1 つです。データソースから情報を解析して取得するために使用されます。REST API に関して言えば、これは `GET` 操作と似ています。GraphQL クエリの主な利点は、アプリケーションの正確なデータ要件を指定して、適切なタイミングで関連データを取得できることです。

データソースにクエリを実行するには

1. まだサインインしていない場合は、 にサインイン AWS Management Console して AppSync コンソールを開きます。
2. テーブルから API を選択します。
3. 左側のタブで [クエリ] を選択します。
4. 表の左側にある [エクスプローラー] タブの `query listTodos` の下で `getTodo` 操作を展開します。



5. コードエディタで、オペレーションコードが表示されます。

```
query listTodos {
  getTodo(id: "") {
    description
```

```
    id
    name
    when
    where
  }
```

(id:"") にミューテーション操作の結果に保存した値を入力します。この例では、次のようになります。

```
query listTodos {
  getTodo(id: "abcdefgh-1234-1234-1234-abcdefghijkl") {
    description
    id
    name
    when
    where
  }
```

6. [実行] を選択し、[ListToDos] を選択します。結果はエディタの右側に表示されます。この URL を次のように表示します。

```
{
  "data": {
    "getTodo": {
      "description": "I need to buy eggs",
      "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
      "name": "Shopping List",
      "when": "Friday",
      "where": "Home"
    }
  }
}
```

> **Note**
>
> クエリは、指定したフィールドのみを返します。不要なフィールドは、リターンフィールドから削除することで選択解除できます。
>
> ```
> {
>     description
>     id
> ```

```
    name
    when
    where
  }
```

[エクスプローラー] タブで削除したいフィールドの横にあるチェックボックスをオフにすることもできます。

7. データソースにエントリを作成する手順を繰り返し、listTodos 操作でクエリ手順を繰り返すことで、listTodos 操作を試すこともできます。2 つ目のタスクを追加した例を次に示します。

```
{
  "createtodoinput": {
    "name": "Second Task",
    "when": "Monday",
    "where": "Home",
    "description": "I need to mow the lawn"
  }
}
```

listTodos 操作を呼び出すと、古いエントリと新しいエントリの両方が返されました。

```
{
  "data": {
    "listTodos": {
      "items": [
        {
          "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
          "name": "Shopping List",
          "when": "Friday",
          "where": "Home",
          "description": "I need to buy eggs"
        },
        {
          "id": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
          "name": "Second Task",
          "when": "Monday",
          "where": "Home",
          "description": "I need to mow the lawn"
        }
      ]
```

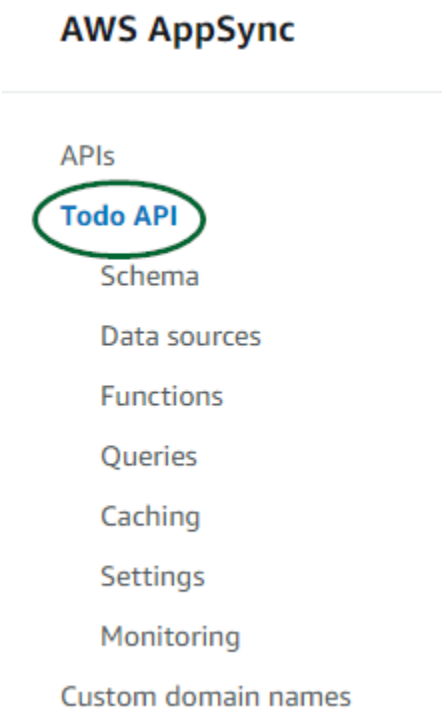
```
    }
  }
}
```

AWS AppSync コンソールの補足セクション

これらのセクションは、より高度な AWS AppSync トピックのリファレンスです。他のことをする前に、「補足資料」セクションを読むことをお勧めします。

Integration

コンソールタブで API の名前を選択すると、統合ページが表示されます。



API を設定するためのステップが説明され、クライアントアプリケーションを作成するための次のステップが記載されています。「アプリとの統合」セクションでは、AWS Amplify ツールチェーンを使用して、設定およびコード生成を通じて API を iOS、Android、および JavaScript アプリケーションと接続するプロセスを自動化する詳細が記載されています。Amplify ツールチェーンでは、チェインのローカルワークステーションからプロジェクトを構築するために至るまで完全にサポートされ、GraphQL のプロビジョニングや CI/CD のワークフローなどのローカルワークステーションからのプロジェクトのビルドを完全にサポートしています。

「クライアントサンプル」セクションには、エンドツーエンドのエクスペリエンスをテストするためのサンプルクライアントアプリケーション (JavaScript、iOS、Android など) も掲載されています。開始するために必要な情報 (エンドポイント URL など) を含む設定ファイルと共に、これらのサンプルをクローンし、ダウンロードできます。「AWS Amplify ツールチェーン」ページの手順に従ってアプリケーションを実行します。

補足資料

- GraphQL API の設計 - これは、データソースやリゾルバーのない空白のスキーマを使用して GraphQL を作成するための包括的なガイドです。

を使用した GraphQL APIsの設計 AWS AppSync

AWS AppSync では、コンソールエクスペリエンスを使用して GraphQL APIsを作成できます。「サンプルスキーマの起動」セクションでこれを垣間見ました。ただし、このガイドでは、活用できるオプションと設定のカタログ全体は表示されていません AWS AppSync。

コンソールで GraphQL API を作成することを選択した場合、検討すべきオプションがいくつかあります。「サンプルスキーマの起動」ガイドを読んだ方には、定義済みのモデルから API を作成する方法を紹介しました。以下のセクションでは、 AWS AppSyncで GraphQL API を作成するためのその他のオプションと構成について説明します。

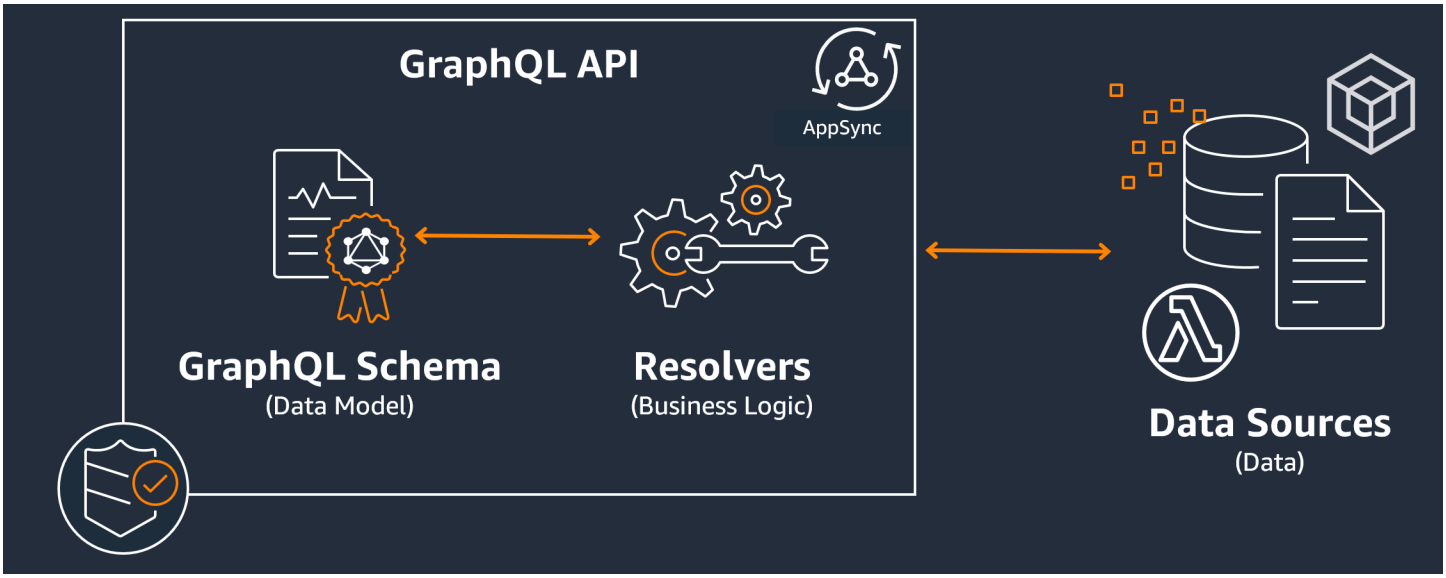
このセクションでは、以下を行います。

1. Blank APIs or imports: このガイドでは、GraphQL API を作成するための作成プロセス全体を説明します。モデルのない空白のテンプレートから GraphQL を作成する方法、スキーマのデータソースを設定する方法、最初のリゾルバーをフィールドに追加する方法を学習します。
2. Real-time data: このガイドでは、 の WebSocket エンジンを使用して API AWS AppSyncを作成するための潜在的なオプションを示します。
3. Merged APIs: このガイドでは、複数の既存の GraphQL API のデータを関連付けてマージすることにより、新しい GraphQL API を作成する方法を説明します。
4. the section called “RDS イントロスペクションを使用した GraphQL API の構築”: このガイドでは、Data API を使用して Amazon RDS テーブルを統合する方法を説明します。

GraphQL API の構築 (空の API またはインポートされた API)

空白のテンプレートから GraphQL API を作成する前に、GraphQL に関する概念を確認しておくと役に立ちます。GraphQL API には次の 3 つの基本的なコンポーネントがあります。

1. スキーマは、データの形状と定義が含まれているファイルです。GraphQL サービスに対してクライアントによってリクエストが実行されるとき、返されるデータはスキーマの仕様に従います。詳細については、「GraphQL スキーマ」を参照してください。
2. データソースはスキーマにアタッチされます。リクエストが実行されると、ここでデータが取得され、変更されます。詳細については、「Data sources」を参照してください。
3. リゾルバーはスキーマとデータソースの間に存在します。リクエストが実行されると、リゾルバーはソースからデータに対してオペレーションを実行し、結果をレスポンスとして返します。詳細については、「Resolvers」を参照してください。



AWS AppSync は、スキーマとリゾルバーのコードを作成、編集、保存できるようにすることで APIs を管理します。データソースは、データベース、DynamoDB テーブル、Lambda 関数などの外部リポジトリから取得されます。 AWS サービスを使用してデータを保存する場合、またはデータを保存することを計画している場合、 は AWS アカウントから GraphQL APIs にデータを関連付けたときに、ほぼシームレスなエクスペリエンス AWS AppSync を提供します。

次のセクションでは、 AWS AppSync サービスを使用してこれらの各コンポーネントを作成する方法について説明します。

トピック

- GraphQL スキーマの設計
- AWS AppSync でのデータソースのアタッチ
- でのリゾルバーの設定 AWS AppSync
- AWS CDK での AWS AppSync API の使用

GraphQL スキーマの設計

GraphQL スキーマは、あらゆる GraphQL サーバー実装の基盤です。各 GraphQL API は、リクエストからのデータがどのように入力されるかを記述するタイプとフィールドを含む単一のスキーマによって定義されます。API を経由するデータフローと実行される操作は、スキーマと照合して検証する必要があります。

GraphQL タイプシステムは、GraphQL サーバーの機能を記述し、クエリが有効であることを判別するために使用されます。サーバーのタイプシステムはしばしばそのサーバーのスキーマと呼ばれ、さ

まざまなオブジェクトタイプ、スカラー型、入力型などで構成されます。GraphQL は宣言型であると同時に厳密に型付けされています。つまり、型はランタイムに明確に定義され、指定されたものだけを返します。

AWS AppSync では、GraphQL スキーマを定義および設定できます。次のセクションでは、 AWS AppSyncのサービスを使用して GraphQL スキーマを最初から作成する方法について説明します。

GraphQL スキーマの構造化

> **Tip**
>
> 続ける前に「スキーマ」セクションを確認することをおすすめします。

GraphQL は API サービスを実装するための強力なツールです。GraphQL のウェブサイトによると、GraphQL は以下の通りです。

「GraphQL は API 用のクエリ言語であり、既存のデータでそれらのクエリを実行するためのランタイムです。GraphQL は、API 内のデータを完全かつわかりやすく説明し、クライアントが必要としているものだけを尋ねることができるようにし、時間の経過とともに API を進化させやすくし、強力な開発者ツールを可能にします」

このセクションでは、GraphQL 実装の最初の部分であるスキーマについて説明します。上の引用を使用すると、スキーマは「API 内のデータを完全かつわかりやすく説明する」役割を果たします。言い換えると、GraphQL スキーマは、サービスのデータ、操作、およびそれらの間の関係をテキストで表現したものです。スキーマは、GraphQL サービス実装の主要なエントリポイントと見なされます。当然のことながら、多くの場合、プロジェクトで最初に作成するものの 1 つです。次に進む前に「スキーマ」セクションを確認することをおすすめします。

「スキーマ」セクションを引用すると、GraphQL スキーマはスキーマ定義言語 (SDL) で記述されています。SDL は、構造が確立された型とフィールドで構成されています。

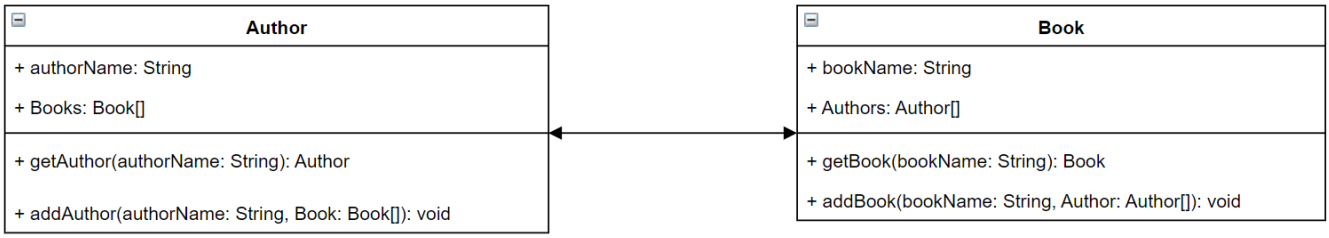
- タイプ: タイプとは、GraphQL がデータの形状と動作を定義する方法です。GraphQL は、このセクションの後半で説明する多数の型をサポートしています。スキーマで定義されている各タイプには、独自のスコープが含まれます。スコープ内には、GraphQL サービスで使用される値または ロジックを含むことができる 1 つ以上のフィールドがあります。型にはさまざまな役割がありますが、最も一般的なのはオブジェクトまたはスカラー (プリミティブ値型) です。
- フィールド: フィールドはタイプのスコープ内に存在し、GraphQL サービスから要求された値を保持します。これらは他のプログラミング言語の変数とよく似ています。フィールドで定義する

データの形状によって、リクエスト/レスポンス操作におけるデータの構造が決まります。これにより、開発者はサービスのバックエンドがどのように実装されているかを知らなくても、何が返されるかを予測できます。

最も単純なスキーマには、次の 3 つの異なるデータカテゴリが含まれます。

1. スキーマルート: ルートはスキーマのエントリポイントを定義します。データの追加、削除、変更などの操作を行うフィールドを指します。
2. タイプ: これらはデータの形状を表すために使用される基本タイプです。これらは概して、定義された特性を持つ何かのオブジェクト、または抽象的な表現と考えることができます。例えば、データベース内の人を表す `Person` オブジェクトを作成できます。各個人の特性はフィールド `Person` として内部で定義されます。名前、年齢、職業、住所など何でも構いません。
3. 特殊オブジェクトタイプ: スキーマ内の操作の動作を定義するタイプです。特殊オブジェクトタイプはそれぞれ、スキーマごとに 1 回定義されます。これらは最初にスキーマのルートに配置され、次にスキーマ本体で定義されます。特別なオブジェクトタイプの各フィールドは、リゾルバーによって実装される単一の操作を定義します。

これを大局的に見ると、著者とその著者が書いた本を保存するサービスを作成していると想像してみてください。各著者には名前と執筆した本の配列があります。各本には名前と関連する著者のリストがあります。また、本や著者を追加したり、検索したりできる機能も必要です。この関係を単純な UML 表現で表現すると、次のようになります。



GraphQL では、`Author` および `Book` エンティティとはスキーマ内の 2 つの異なるオブジェクトタイプを表します。

```
type Author {
}

type Book {
```

```
}
```

Author には authorName と Books が含まれ、Book には bookName とAuthors が含まれます。これらはタイプのスコープ内のフィールドとして表すことができます。

```
type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}
```

ご覧のとおり、型表現は図と非常に似ています。しかし、メソッドは少し難しいところです。これらは、いくつかの特殊なオブジェクトタイプのいずれかにフィールドとして配置されます。特殊なオブジェクト分類は、その動作によって異なります。GraphQL には、クエリ、ミューテーション、およびサブスクリプションの 3 つの基本的な特殊オブジェクトタイプがあります。特別なオブジェクトの詳細については、 を参照してください。

getAuthorとgetBookはどちらもデータを要求しているため、Query の特別なオブジェクトタイプに配置されます。

```
type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}

type Query {
  getAuthor(authorName: String): Author
  getBook(bookName: String): Book
}
```

オペレーションはクエリにリンクされ、クエリ自体もスキーマにリンクされます。スキーマルートを追加すると、特別なオブジェクトタイプ (この場合は Query) がエントリポイントの 1 つとして定義されます。これは schema キーワードを使用して行うことが D けいます。

```
schema {
  query: Query
}

type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}

type Query {
  getAuthor(authorName: String): Author
  getBook(bookName: String): Book
}
```

最後の 2 つの方法を見てみると、addAuthor と addBook はデータベースにデータを追加しているので、これらはMutation の特別なオブジェクトタイプで定義されます。ただし、タイプページを見ると、オブジェクトを直接参照する入力は厳密には出力タイプなので、許可されていないこともわかります。この場合、Author または Book は使用できないため、同じフィールドを持つ入力タイプを作成する必要があります。この例では、AuthorInput と BookInput を追加しましたが、どちらもそれぞれのタイプの同じフィールドを受け入れます。次に、入力をパラメータとして使用してミューテーションを作成します。

```
schema {
  query: Query
  mutation: Mutation
}

type Author {
  authorName: String
  Books: [Book]
}
```

```
input AuthorInput {
  authorName: String
  Books: [BookInput]
}

type Book {
  bookName: String
  Authors: [Author]
}

input BookInput {
  bookName: String
  Authors: [AuthorInput]
}

type Query {
  getAuthor(authorName: String): Author
  getBook(bookName: String): Book
}

type Mutation {
  addAuthor(input: [BookInput]): Author
  addBook(input: [AuthorInput]): Book
}
```

先ほど行ったことを振り返ってみましょう。

1. エンティティを表すための Book および Author タイプを含むスキーマを作成しました。
2. エンティティの特徴を含むフィールドを追加しました。
3. この情報をデータベースから取得するクエリを追加しました。
4. データベース内のデータを操作するミューテーションを追加しました。
5. GraphQL のルールに準拠するため、ミューテーション内のオブジェクトパラメータを置き換える入力タイプを追加しました。
6. GraphQL 実装がルートタイプの場所を理解できるように、クエリとミューテーションをルートスキーマに追加しました。

ご覧のとおり、スキーマを作成するプロセスには、一般的なデータモデリング (特にデータベースモデリング) から多くの概念が取り入れられています。スキーマはソースからのデータの形に合っていると考えることができます。リゾルバーが実装するモデルとしても機能します。以下のセクションで

は、さまざまな AWSバックアップツールとサービスを使用してスキーマを作成する方法について説明します。

> **Note**
>
> 以下のセクションの例は、実際のアプリケーションで実行するためのものではありません。これらの説明は、ユーザーが独自のアプリケーションを構築できるようにコマンドを紹介することのみを目的としています。

スキーマの作成

スキーマは というファイルにありますschema.graphql。 AWS AppSync を使用すると、ユーザーはさまざまな方法を使用して GraphQL APIs の新しいスキーマを作成できます。この例では、空の API と空白のスキーマを作成します。

Console

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   a. ダッシュボードで、[API の作成] を選択します。
   b. [API オプション] で、[GraphQL API]、[最初から設計]、[次へ] の順に選択します。
      i. API 名では、あらかじめ入力されている名前をアプリケーションに必要な名前に変更します。
      ii. 連絡先情報については、API のマネージャーを特定する連絡先を入力できます。これはオプションのフィールドです。
      iii. [Private API 設定] で、プライベート API 機能を有効にできます。プライベート API には、設定された VPC エンドポイント (VPCE) からのみアクセスできます。詳細については、「プライベート DNS」を参照してください。

      この例では、この機能を有効にすることはお勧めしません。[次へ] 選択して入力を確認します。
   c. [GraphQL タイプを作成] では、データソースとして使用する DynamoDB テーブルを作成するか、この作業をスキップして後で作成するかを選択できます。

      この例では、[GraphQL リソースを後で作成] を選択します。リソースは別のセクションで作成します。

d. 入力内容を確認し、[API を作成] を選択します。

2. 特定の API のダッシュボードが表示されます。API の名前がダッシュボードの上部にあるのでわかります。そうでない場合は、サイドバーで [API] を選択し、API ダッシュボードで API を選択できます。

   - API 名の下のサイドバーで、[スキーマ] を選択します。

3. スキーマエディタでは、ファイルを設定できます。`schema.graphql`ファイルは空でも、モデルから生成されたタイプで埋め込まれていてもかまいません。右側には、リゾルバーをスキーマフィールドにアタッチするための [リゾルバー] セクションがあります。このセクションではリゾルバーについては説明しません。

CLI

> **Note**
>
> CLI を使用するときは、サービス内のリソースにアクセスして作成するための正しい権限を持っていることを確認してください。サービスにアクセスする必要がある管理者以外のユーザーには、最小特権ポリシーを設定するとよいでしょう。 AWS AppSync ポリシーの詳細については、「 の Identity and Access Management AWS AppSync」を参照してください。
> まだの場合は、コンソール版最初に読むことをお勧めします。

1. まだをインストールしてしていない場合は、 AWS CLI をインストールして設定します。
2. `create-graphql-api`コマンドを実行して GraphQL API オブジェクトを作成します。

   この特定のコマンドには、次の 2 つのパラメータを入力する必要があります。

   1. API の `name` です。
   2. `authentication-type`、または API へのアクセスに使用される認証情報の種類 (IAM、OIDC など)。

   > **Note**
   >
   > `Region` など、その他のパラメータは設定する必要がありますが、通常はデフォルトで CLI 設定値になります。

コマンドの例は、次のようになります。

```
aws appsync create-graphql-api --name testAPI123 --authentication-type API_KEY
```

出力は CLI に返されます。例を示します。

```
{
    "graphqlApi": {
        "xrayEnabled": false,
        "name": "testAPI123",
        "authenticationType": "API_KEY",
        "tags": {},
        "apiId": "abcdefghijklmnopqrstuvwxyz",
        "uris": {
            "GRAPHQL": "https://zyxwvutsrqponmlkjihgfedcba.appsync-api.us-
west-2.amazonaws.com/graphql",
            "REALTIME": "wss://zyxwvutsrqponmlkjihgfedcba.appsync-realtime-
api.us-west-2.amazonaws.com/graphql"
        },
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz"
    }
}
```

3.

> **Note**
>
> これはオプションコマンドで、既存のスキーマを取得し、base-64 BLOB を使用して AWS AppSync サービスにアップロードします。この例では、このコマンドは使用しません。

`start-schema-creation` コマンドを実行します。

この特定のコマンドには、次の 2 つのパラメータを入力する必要があります。

1. `api-id` は前のステップからのものです。
2. スキーマ `definition`は base-64 でエンコードされたバイナリブロブです。

コマンドの例は、次のようになります。

```
 aws appsync start-schema-creation --api-id abcdefghijklmnopqrstuvwxyz --
definition "aa1111aa-123b-2bb2-c321-12hgg76cc33v"
```

次のような出力が返されます。

```
{
    "status": "PROCESSING"
}
```

このコマンドは処理後の最終出力を返しません。結果を確認するには別のコマンド `get-schema-creation-status` を使用する必要があります。この 2 つのコマンドは非同期なので、スキーマの作成中でも出力ステータスを確認できることに注意してください。

CDK

> **Tip**
>
> CDK を使用する前に、CDK の公式ドキュメントと CDK AWS AppSyncリファレンスを確認することをお勧めします。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html
> 以下の手順では、特定のリソースを追加するために使用するスニペットの一般的な例のみを示しています。これは本番稼働用コードで機能するソリューションとなることを意図したものではありません。また、動作するアプリが既にあることを前提としています。

1. CDK の開始点は少し異なります。理想的には、`schema.graphql` ファイルはすでに作成されているはずです。必要なのは、`.graphql` ファイル拡張子が付いた新しいファイルを作成することだけです。空のリストを指定できます。
2. 一般的には、使用しているサービスにインポートディレクティブを追加しなければならない場合があります。たとえば、次の形式に従います。

```
import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'
import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
```

GraphQL API を追加するには、スタックファイルで AWS AppSync サービスをインポートする必要があります。

```
import * as appsync from 'aws-cdk-lib/aws-appsync';
```

> **Note**
>
> つまり、appsync キーワードでサービス全体をインポートすることになります。アプリでこれを使用するには、 AWS AppSync コンストラクトは の形式を使用しますappsync.construct_name。例えば、GraphQL API を作りたいなら、new appsync.GraphqlApi(args_go_here) と言うでしょう。次のステップはこれを描写しています。

3. 最も基本的な GraphQL API には、API 用の name と schema パスが含まれます。

```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {
  name: 'name_of_API_in_console',
  schema: appsync.SchemaFile.fromAsset(path.join(__dirname,
 'schema_name.graphql')),
});
```

> **Note**
>
> このスニペットが何をするのか見てみましょう。api の範囲内では、appsync.GraphqlApi(scope: Construct, id: string, props: GraphqlApiProps) を呼び出して新しい GraphQL API を作成しています。スコープは this で、現在のオブジェクトを参照します。ID は *API_ID* で、作成 AWS CloudFormation 時に GraphQL API のリソース名になります。GraphqlApiProps には、GraphQL API の name と schema が含まれています。schema は、.graphql ファイル (*schema_name.graphql*) の絶対パス (__dirname) を検索してスキーマ (SchemaFile.fromAsset) を生成します。実際のシナリオでは、スキーマファイルはおそらく CDK アプリ内にあるでしょう。GraphQL API に加えた変更を使用するには、アプリを再デプロイする必要があります。

スキーマへのタイプの追加

スキーマを追加したので、入力タイプと出力タイプの両方を追加し始めることができます。ここで説明する型は実際のコードでは使用しないでください。これらはプロセスの理解に役立つ例に過ぎません。

まず、オブジェクトタイプを作成します。実際のコードでは、これらのタイプから始める必要はありません。GraphQL の規則と構文に従っていれば、いつでも好きなタイプを作ることができます。

> **Note**
>
> 次のいくつかのセクションではスキーマエディターを使用するので、これを開いたままにしておきます。

Console

- オブジェクトタイプは、type キーワードとタイプ名を使用して作成できます。

```
type Type_Name_Goes_Here {}
```

タイプのスコープ内には、オブジェクトの特性を表すフィールドを追加できます。

```
type Type_Name_Goes_Here {
  # Add fields here
}
```

例を示します。

```
type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}
```

> **Note**
>
> このステップでは、必須の id フィールドを ID として格納し、title フィールドを String として格納し、date フィールドを AWSDateTime として格納する汎用オブ

ジェクトタイプを追加しました。タイプとフィールド、およびその機能の一覧については、「スキーマ」を参照してください。スカラーの一覧とその機能については、タイプのリファレンスを参照してください。

CLI

> **Note**
>
> まだの場合は、コンソール版最初に読むことをお勧めします。

- オブジェクトタイプは `create-type` コマンドを実行して作成できます。

  この特定のコマンドでは、いくつかのパラメータを入力する必要があります。

  1. API の `api-id` です。
  2. `definition`、またはタイプのコンテンツです。コンソールの例では、次のようになっていました。

     ```
     type Obj_Type_1 {
       id: ID!
       title: String
       date: AWSDateTime
     }
     ```

  3. 入力の `format` です。この例では、SDL を使用します。

  コマンドの例は、次のようになります。

  ```
  aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type
   Obj_Type_1{id: ID! title: String date: AWSDateTime}" --format SDL
  ```

  出力は CLI に返されます。例を示します。

  ```
  {
      "type": {
          "definition": "type Obj_Type_1{id: ID! title: String date:
   AWSDateTime}",
  ```

```
        "name": "Obj_Type_1",
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Obj_Type_1",
        "format": "SDL"
    }
}
```

> **Note**
>
> このステップでは、必須の id フィールドを ID として格納し、title フィールドを String として格納し、date フィールドを AWSDateTime として格納する汎用オブジェクトタイプを追加しました。タイプとフィールド、およびその機能の一覧については、「スキーマ」を参照してください。スカラーの一覧とその機能については、タイプのリファレンスを参照してください。
> さらにお気づきかもしれませんが、定義を直接入力しても、小さいタイプでは有効ですが、大きい型や複数のタイプを追加することはできません。.graphql ファイルにすべてを追加し、それを入力として渡すこともできます。

CDK

> **Tip**
>
> CDK を使用する前に、CDK の公式ドキュメントと CDK AWS AppSyncリファレンスを確認することをお勧めします。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html
> 以下の手順では、特定のリソースを追加するために使用するスニペットの一般的な例のみを示しています。これは本番稼働用コードで機能するソリューションとなることを意図したものではありません。また、動作するアプリが既にあることを前提としています。

タイプを追加するには、.graphql ファイルに追加する必要があります。例えば、コンソールの例は次のとおりです。

```
type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
```

```
}
```

他のファイルと同様に、タイプをスキーマに直接追加できます。

> **Note**
>
> GraphQL API に加えた変更を使用するには、アプリを再デプロイする必要があります。

オブジェクトタイプには、文字列や整数などのスカラー型であるフィールドがあります。 AWS AppSync では、基本 GraphQL スカラーAWSDateTimeに加えて、 などの拡張スカラー型を使用することもできます。また、名前が感嘆符で終わっているフィールドは必須フィールドです。

ID スカラー型は、String または Int のいずれかである一意の識別子です。これらはリゾルバーのマッピングテンプレートで自動割り当てを制御できます。

Query のような特別なオブジェクトタイプと上記の例のような「通常の」オブジェクトタイプでは、どちらもtype キーワードを使用し、オブジェクトと見なされるという点で類似点があります。ただし、特殊オブジェクトタイプ (Query、Mutation、およびSubscription) は API のエントリポイントとして公開されるため、動作が大きく異なります。また、データというよりは、オペレーションのシェーピングに関するものでもあります。詳細については、「The Query and Mutation types」を参照してください。

特殊なオブジェクトタイプについて言えば、次のステップは、1 つまたは複数のオブジェクトタイプを追加して、シェーピングされたデータに対して操作を実行することです。実際のシナリオでは、すべての GraphQL スキーマには、データをリクエストするためのルートクエリタイプが少なくとも必要です。クエリは、GraphQL サーバーのエントリポイント (またはエンドポイント) の 1 つと考えることができます。例としてクエリを追加してみましょう。

Console

- クエリを作成するには、他のタイプと同様にスキーマファイルに追加するだけです。クエリには、次のような Query タイプとルート内のエントリが必要です。

```
schema {
  query: Name_of_Query
}

type Name_of_Query {
  # Add field operation here
```

```
}
```

プロダクション環境の *Name_of_Query* は、ほとんどの場合、単純に Query と呼ばれることに注意してください。この値のままにしておくことをお勧めします。クエリタイプ内にはフィールドを追加できます。各フィールドはリクエスト内でオペレーションを実行します。その結果、すべてではないにしても、ほとんどのフィールドがリゾルバーにアタッチされます。ただし、このセクションではその点については触れません。フィールドのオペレーションの形式については、次のようになります。

```
Name_of_Query(params): Return_Type # version with params
Name_of_Query: Return_Type # version without params
```

例を示します。

```
schema {
  query: Query
}

type Query {
  getObj: [Obj_Type_1]
}

type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}
```

> **Note**
>
> このステップでは、Query タイプを追加して schema ルートで定義しました。Query タイプは、Obj_Type_1 オブジェクトのリストを返す getObj フィールドを定義しました。Obj_Type_1 が前のステップのオブジェクトであることに注意してください。プロダクションコードでは、フィールド操作は通常、Obj_Type_1 のようなオブジェクトによって形成されたデータを処理します。さらに、getObj のようなフィールドには通常、ビジネスロジックを実行するリゾルバーがあります。これについては別のセクションで説明します。
>
> 追加の注意点として、 はエクスポート中にスキーマルート AWS AppSync を自動的に追加するため、技術的にはスキーマに直接追加する必要はありません。弊社のサー

ビスは重複するスキーマを自動的に処理します。ベストプラクティスとしてここに追加しています。

CLI

> **Note**
> まだの場合は、コンソール版最初に読むことをお勧めします。

1. create-type コマンドを実行して query 定義付きの schema ルートを作成します。

   この特定のコマンドでは、いくつかのパラメータを入力する必要があります。

   1. API の api-id です。
   2. definition、またはタイプのコンテンツです。コンソールの例では、次のようになっていました。

      ```
      schema {
        query: Query
      }
      ```

   3. 入力の format です。この例では、SDL を使用します。

   コマンドの例は、次のようになります。

   ```
   aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "schema
    {query: Query}" --format SDL
   ```

   出力は CLI に返されます。例を示します。

   ```
   {
       "type": {
           "definition": "schema {query: Query}",
           "name": "schema",
           "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
   abcdefghijklmnopqrstuvwxyz/types/schema",
           "format": "SDL"
   ```

```
    }
}
```

> **Note**
>
> create-type コマンドに正しく入力しなかった場合は、update-type コマンドを実行することでスキーマルート (またはスキーマ内の任意のタイプ) を更新できることに注意してください。この例では、subscription 定義を含むようにスキーマルートを一時的に変更します。
> この特定のコマンドでは、いくつかのパラメータを入力する必要があります。
>
> 1. API の api-id です。
> 2. タイプの type-name。コンソールの例では、これは schema でした。
> 3. definition、またはタイプのコンテンツです。コンソールの例では、次のようになっていました。
>
>    ```
>    schema {
>      query: Query
>    }
>    ```
>
>   subscription を追加した後のスキーマは次のようになります。
>
>    ```
>    schema {
>      query: Query
>      subscription: Subscription
>    }
>    ```
> 4. 入力の format です。この例では、SDL を使用します。
>
> コマンドの例は、次のようになります。
>
> ```
> aws appsync update-type --api-id abcdefghijklmnopqrstuvwxyz --type-name
>  schema --definition "schema {query: Query subscription: Subscription}"
>  --format SDL
> ```
>
> 出力は CLI に返されます。例を示します。
>
> ```
> {
> ```

```
    "type": {
        "definition": "schema {query: Query subscription: Subscription}",
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/schema",
        "format": "SDL"
    }
}
```

この例でも、フォーマット済みのファイルを追加しても問題ありません。

2. create-type コマンドを実行して Query タイプを作成します。

この特定のコマンドでは、いくつかのパラメータを入力する必要があります。

1. API の api-id です。
2. definition、またはタイプのコンテンツです。コンソールの例では、次のようになっていました。

```
type Query {
  getObj: [Obj_Type_1]
}
```

3. 入力の format です。この例では、SDL を使用します。

コマンドの例は、次のようになります。

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type
 Query {getObj: [Obj_Type_1]}" --format SDL
```

出力は CLI に返されます。例を示します。

```
{
    "type": {
        "definition": "Query {getObj: [Obj_Type_1]}",
        "name": "Query",
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Query",
        "format": "SDL"
    }
}
```

> **Note**
>
> このステップでは、Query タイプを追加して schema ルートで定義しました。Query タイプは、Obj_Type_1 オブジェクトのリストを返す getObj フィールドを定義しました。
> schema ルートコードではquery: Query、query: 部分はクエリがスキーマで定義されたことを示し、Query 部分は実際の特別なオブジェクト名を示します。

CDK

> **Tip**
>
> CDK を使用する前に、CDK の公式ドキュメントと CDK AWS AppSyncリファレンスを確認することをお勧めします。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html
> 以下の手順では、特定のリソースを追加するために使用するスニペットの一般的な例のみを示しています。これは本番稼働用コードで機能するソリューションとなることを意図したものではありません。また、動作するアプリが既にあることを前提としています。

クエリとスキーマルートを .graphql ファイルに追加する必要があります。この例は以下の例のようになっていますが、実際のスキーマコードに置き換える必要があります。

```
schema {
  query: Query
}

type Query {
  getObj: [Obj_Type_1]
}

type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}
```

他のファイルと同様に、タイプをスキーマに直接追加できます。

> **Note**
>
> スキーマルートの更新は任意です。ベストプラクティスとして、この例に追加しました。GraphQL API に加えた変更を使用するには、アプリを再デプロイする必要があります。

オブジェクトと特殊オブジェクト (クエリ) の両方を作成する例を見てきました。また、これらを相互に接続してデータや操作を記述する方法についても説明しました。データ記述と 1 つ以上のクエリのみを含むスキーマを作成できます。ただし、データソースにデータを追加する操作をもう 1 つ追加したいと考えています。データを変更する Mutation と呼ばれる特別なオブジェクトタイプを追加します。

Console

- ミューテーションはMutationと呼ばれます。Query と同様に、Mutation 内部のフィールド操作は操作を記述し、リゾルバーにアタッチされます。また、これは特殊なオブジェクトタイプなので、schema ルートで定義する必要があることにも注意してください。ミューテーションの例を示します。

```
schema {
  mutation: Name_of_Mutation
}

type Name_of_Mutation {
  # Add field operation here
}
```

一般的なミューテーションは、クエリのようにルートにリストされます。ミューテーションは type キーワードと名前を使って定義されます。通常は *Name_of_Mutation* がMutationと呼ばれるので、そのままにしておくことをおすすめします。各フィールドはオペレーションも実行します。フィールドのオペレーションの形式については、次のようになります。

```
Name_of_Mutation(params): Return_Type # version with params
Name_of_Mutation: Return_Type # version without params
```

例を示します。

```
schema {
  query: Query
  mutation: Mutation
}

type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}

type Query {
  getObj: [Obj_Type_1]
}

type Mutation {
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
}
```

Note

このステップでは、addObj フィールドのある Mutation タイプを追加しました。このフィールドの機能をまとめてみましょう。

```
addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
```

addObj では Obj_Type_1 オブジェクトを使用してオペレーションを実行しています。これはフィールドがあるから明らかですが、構文では : Obj_Type_1 戻り値のタイプでもそれが証明されています。addObj 内部では、Obj_Type_1 オブジェクトのid、title、date フィールドをパラメータとして受け付けています。ご覧のとおり、これはメソッド宣言によく似ています。ただし、メソッドの動作についてはまだ説明していません。前述のように、スキーマはデータとオペレーションの内容を定義するためだけのもので、オペレーションの方法を定義するものではありません。実際のビジネスロジックの実装は、後で最初のリゾルバーを作成するときに実装します。

スキーマが完成したら、それを schema.graphql ファイルとしてエクスポートするオプションがあります。スキーマエディターで [スキーマをエクスポート] を選択すると、サポートされている形式でファイルをダウンロードできます。

追加の注意点として、 はエクスポート中にスキーマルート AWS AppSync を自動的に追加するため、技術的にはスキーマに直接追加する必要はありません。弊社のサービスは重複するスキーマを自動的に処理します。ベストプラクティスとしてここに追加しています。

CLI

> **Note**
>
> まだの場合は、コンソール版最初に読むことをお勧めします。

1. update-type コマンドを実行してルートスキーマを更新します。

   この特定のコマンドでは、いくつかのパラメータを入力する必要があります。

   1. API の api-id です。
   2. タイプの type-name。コンソールの例では、これは schema でした。
   3. definition、またはタイプのコンテンツです。コンソールの例では、次のようになっていました。

      ```
      schema {
        query: Query
        mutation: Mutation
      }
      ```

   4. 入力の format です。この例では、SDL を使用します。

   コマンドの例は、次のようになります。

   ```
   aws appsync update-type --api-id abcdefghijklmnopqrstuvwxyz --type-name schema
    --definition "schema {query: Query mutation: Mutation}" --format SDL
   ```

   出力は CLI に返されます。例を示します。

   ```
   {
       "type": {
           "definition": "schema {query: Query mutation: Mutation}",
   ```

```
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/schema",
        "format": "SDL"
    }
}
```

2. create-type コマンドを実行して Mutation タイプを作成します。

   この特定のコマンドでは、いくつかのパラメータを入力する必要があります。

   1. API の api-id です。
   2. definition、またはタイプのコンテンツです。コンソールの例では、次のようになっていました。

      ```
      type Mutation {
        addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
      }
      ```

   3. 入力の format です。この例では、SDL を使用します。

   コマンドの例は、次のようになります。

   ```
   aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type
    Mutation {addObj(id: ID! title: String date: AWSDateTime): Obj_Type_1}" --
   format SDL
   ```

   出力は CLI に返されます。例を示します。

   ```
   {
       "type": {
           "definition": "type Mutation {addObj(id: ID! title: String date:
    AWSDateTime): Obj_Type_1}",
           "name": "Mutation",
           "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
   abcdefghijklmnopqrstuvwxyz/types/Mutation",
           "format": "SDL"
       }
   }
   ```

CDK

> **Tip**
>
> CDK を使用する前に、CDK の公式ドキュメントと CDK AWS AppSyncリファレンスを確認することをお勧めします。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html
> 以下の手順では、特定のリソースを追加するために使用するスニペットの一般的な例のみを示しています。これは本番稼働用コードで機能するソリューションとなることを意図したものではありません。また、動作するアプリが既にあることを前提としています。

クエリとスキーマルートを `.graphql` ファイルに追加する必要があります。この例は以下の例のようになっていますが、実際のスキーマコードに置き換える必要があります。

```
schema {
  query: Query
  mutation: Mutation
}

type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}

type Query {
  getObj: [Obj_Type_1]
}

type Mutation {
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
}
```

> **Note**
>
> スキーマルートの更新は任意です。ベストプラクティスとして、この例に追加しました。GraphQL API に加えた変更を使用するには、アプリを再デプロイする必要があります。

その他の考慮事項 - 列挙型をステータスとして使用する

これで、基本的なスキーマの作り方がわかりました。しかし、スキーマの機能を高めるために追加できることはたくさんあります。アプリケーションによく見られることの 1 つは、列挙型をステータスとして使用することです。列挙型を使用すると、呼び出し時に値のセットから特定の値を強制的に選択させることができます。これは、長期間にわたって大幅に変化することはないとわかっている場合に便利です。仮説的に言えば、レスポンスにステータスコードまたは文字列を返す列挙型を追加できるかもしれません。

例として、ユーザーの投稿データをバックエンドに保存するソーシャルメディアアプリを作っているとしましょう。このスキーマには、個々の投稿のデータを表す Post タイプが含まれています。

```
type Post {
  id: ID!
  title: String
  date: AWSDateTime
  poststatus: PostStatus
}
```

私たちの Post には一意の id、投稿 title、投稿の date と、アプリによって処理された投稿の状態を表す PostStatus と呼ばれるという列挙型が含まれます。今回の操作では、すべての投稿データを返すクエリを用意します。

```
type Query {
  getPosts: [Post]
}
```

また、データソースに投稿を追加するミューテーションもあります。

```
type Mutation {
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post
}
```

スキーマを見ると、PostStatus 列挙型には複数のステータスがある可能性があります。success (投稿は正常に処理されました)、pending (投稿は処理中)、error (投稿は処理できません) という 3 つの基本的な状態が必要な場合があります。列挙型を追加するには、次のようにします。

```
enum PostStatus {
  success
```

```
  pending
  error
}
```

完全なスキーマは次のようになります。

```
schema {
  query: Query
  mutation: Mutation
}

type Post {
  id: ID!
  title: String
  date: AWSDateTime
  poststatus: PostStatus
}

type Mutation {
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post
}

type Query {
  getPosts: [Post]
}

enum PostStatus {
  success
  pending
  error
}
```

ユーザーがアプリケーションに Post を追加すると、そのデータを処理する addPost オペレーションが呼び出されます。addPost にアタッチされたリゾルバーがデータを処理している間、poststatus はオペレーションの状態で継続的に更新されます。クエリを実行すると、Post にはデータの最終ステータスが含まれます。データをスキーマ内でどのように動作させたいかを説明しているだけだということを覚えておいてください。ここではリゾルバーの実装について多くのことを想定しています。リゾルバーは、リクエストを満たすためにデータを処理するための実際のビジネスロジックを実装します。

オプションとしての考慮事項 - サブスクリプション

AWS AppSync のサブスクリプションは、ミューテーションへのレスポンスとして呼び出されます。サブスクリプションは、Subscription 型と @aws_subscribe( ) ディレクティブを使用してスキーマで設定して、どのミューテーションが 1 つまたは複数のサブスクリプションを呼び出すかを示します。サブスクリプションの設定については、「リアルタイムデータ」を参照してください。

オプションとしての考慮事項 - リレーションとページ分割

DynamoDB テーブルに 100 万個の Posts が格納されていて、そのデータの一部を返したいとします。ただし、上記のクエリ例では、すべての投稿を返すだけです。リクエストするたびにこれらすべてを取得したいとは思わないでしょう。代わりに、それらをページ分割したいと思うかもしれません。スキーマを次のように変更します。

- getPosts フィールドに、nextToken (イテレーター) と limit (イテレーション制限) の 2 つの入力引数を追加します。
- Posts (Post オブジェクトのリストを取得) と nextToken (イテレーター) PostIterator フィールドを含む新しい タイプを追加します。
- getPosts を変更すると、Post オブジェクトのリストではなく、PostIterator を返します。

```
schema {
  query: Query
  mutation: Mutation
}

type Post {
  id: ID!
  title: String
  date: AWSDateTime
  poststatus: PostStatus
}

type Mutation {
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post
}

type Query {
  getPosts(limit: Int, nextToken: String): PostIterator
}
```

```
enum PostStatus {
  success
  pending
  error
}

type PostIterator {
  posts: [Post]
  nextToken: String
}
```

PostIterator タイプにより、Post オブジェクトのリストと、次ののバッチを取得するための nextToken を返すことができます。PostIterator 内部には、ページ分割トークン (nextToken) とともに返される Post アイテム ([Post]) のリストがあります。 AWS AppSync では、これはリゾルバーを介して Amazon DynamoDB に接続され、暗号化されたトークンとして自動的に生成されます。マッピングテンプレートにより、limit 引数の値は maxResults パラメータに変換され、nextToken 引数の値は exclusiveStartKey パラメータに変換されます。 AWS AppSync コンソールでの例および組み込みテンプレートサンプルについては、「リゾルバーのリファレンス (JavaScript)」を参照してください。

AWS AppSync でのデータソースのアタッチ

データソースは、GraphQL APIsできる AWS アカウントのリソースです。 AWS AppSync は、Amazon DynamoDB AWS Lambda、リレーショナルデータベース (Amazon Aurora Serverless)、Amazon OpenSearch Service、HTTP エンドポイントなどの多数のデータソースをサポートしています。 AWS AppSync API は、複数のデータソースとやり取りするように設定できるため、データを 1 つの場所に集約できます。 AWS AppSync は、アカウントから既存の AWS リソースを使用するか、スキーマ定義からユーザーに代わって DynamoDB テーブルをプロビジョニングできます。

次のセクションでは、GraphQL API にデータソースをアタッチする方法について説明します。

データソースのタイプ

AWS AppSync コンソールでスキーマを作成したら、それにデータソースをアタッチできます。API を初めて作成する場合、定義済みスキーマの作成中に Amazon DynamoDB テーブルをプロビジョニングするオプションがあります。ただし、このセクションではそのオプションについては説明しません。この例については、「Launching a schema」セクションを参照してください。

代わりに、 AWS AppSync がサポートするすべてのデータソースを確認します。アプリケーションに適したソリューションを選ぶ要因は多数あります。以下のセクションでは、各データソースの追加のコンテキストについて説明します。データソースに関する一般的な情報については、「データソース」を参照してください。

Amazon DynamoDB

Amazon DynamoDB は、スケーラブルなアプリケーション向けの AWS主要なストレージソリューションの 1 つです。DynamoDB のコアコンポーネントは テーブルで、これは単なるデータのコレクションです。通常テーブルは、`Book` や `Author` などのエンティティに基づいて作成されます。テーブルエントリの情報は、各エントリに固有のフィールドのグループであるアイテムとして保存されます。アイテム全体は、データベース内の 1 つの行またはレコードを表します。たとえば、`Book` エントリのアイテムにはその値とともに `title` と `author` が含まれる場合があります。`title` や `author` などの個々のフィールドは属性と呼ばれ、リレーショナルデータベースの列の値に似ています。

ご想像のとおり、テーブルはアプリケーションからのデータの保存に使用されます。 AWS AppSync を使用すると、DynamoDB テーブルを GraphQL API に接続してデータを操作できます。フロントエンドのウェブとモバイルのブログからこのユースケースを見てみましょう。このアプリケーションでは、ユーザーがソーシャルメディアアプリにサインアップできます。ユーザーはグループに参加して投稿をアップロードできます。この投稿は、そのグループに登録している他のユーザーにブロードキャストされます。ユーザーのアプリケーションは、ユーザー、投稿、ユーザーグループの情報を DynamoDB に保存します。GraphQL API (によって管理 AWS AppSync) は DynamoDB テーブルとインターフェイスします。フロントエンドに反映される変更をユーザーがシステムで行うと、GraphQL API はその変更を取得し、リアルタイムで他のユーザーにブロードキャストします。

AWS Lambda

Lambda は、イベントへの応答としてコードを実行するために必要なリソースを自動的に構築するイベント駆動型サービスです。Lambda は、リソースを実行するためのコード、依存関係、設定を含むグループステートメントである関数を使用します。関数は、関数を呼び出すアクティビティのグループであるトリガーを検出すると自動的に実行されます。トリガーは、API コールを行うアプリケーション、リソースをスピンアップするアカウントの AWS サービスなどです。トリガーされると、関数は、変更するデータを含む JSON ドキュメントであるイベントを処理します。

Lambda は、コードを実行するためのリソースをプロビジョニングしなくてもコードを実行するのに適しています。フロントエンドのウェブとモバイルのブログからこのユースケースを見てみましょう。このユースケースは、DynamoDB セクションで紹介した内容と少し似ています。このアプリケーションでは、GraphQL API により投稿の追加 (ミューテーション) やそのデータの取得 (ク

エリ) などのオペレーションが定義されます。オペレーション (`getPost ( id: String ! ) : Post`、`getPostsByAuthor ( author: String ! ) : [ Post ]` など) の機能を実装するために、Lambda 関数が使用され、インバウンドリクエストが処理されます。オプション 2: Lambda リゾルバー AWS AppSync では、 AWS AppSync サービスを使用してスキーマを維持し、Lambda データソースをオペレーションの 1 つにリンクします。オペレーションが呼び出されると、Lambda は Amazon RDS Proxy とやり取りして、データベース上でビジネスロジックを実行します。

Amazon RDS

Amazon RDS では、リレーショナルデータベースを迅速に構築して設定できます。Amazon RDS では、クラウド内の隔離されたデータベース環境として機能する汎用データベースインスタンスを作成します。このインスタンスでは、実際の RDBMS ソフトウェア (PostgreSQL、MySQL など) である DB エンジンを使用します。このサービスは、 AWSインフラストラクチャ、パッチ適用や暗号化などのセキュリティサービスを使用してスケーラビリティを提供し、デプロイの管理コストを削減することで、バックエンド作業の多くをオフロードします。

Lambda セクションの同じユースケースを見てみましょう。オプション 3:Amazon RDS リゾルバー AWS AppSync では、 の GraphQL API を Amazon RDS に直接リンク AWS AppSync する別のオプションがあります。このオプションでは、データ API を使用して、データベースが GraphQL API に関連付けられます。リゾルバーはフィールド (通常はクエリ、ミューテーション、サブスクリプション) にアタッチされ、データベースへのアクセスに必要な SQL ステートメントを実装します。クライアントによってフィールドを呼び出すリクエストが行われると、リゾルバーはステートメントを実行してレスポンスを返します。

Amazon EventBridge

EventBridge では、アタッチするサービスまたはアプリケーション (イベントソース) からイベントを受信し、一連のルールに基づいて処理するパイプラインであるイベントバスを作成します。イベントは実行環境における一種の状態の変化であり、ルールはイベントのフィルタのセットです。ルールは、イベントパターン、すなわちイベントの状態変化のメタデータ (ID、リージョン、アカウント番号、ARN など) に従います。イベントがイベントパターンと一致すると、EventBridge はパイプラインを介して送信先サービス (ターゲット) にイベントを送信し、ルールで指定されたアクションをトリガーします。

EventBridge は、状態が変化するオペレーションを他のサービスにルーティングするのに適しています。フロントエンドのウェブとモバイルのブログからこのユースケースを見てみましょう。この例は、異なるサービスを複数のチームで管理している e コマースソリューションを示しています。これらのサービスの 1 つは、フロントエンドの配送の各ステップ (注文済み、進行中、出荷済み、配送済みなど) で顧客に注文の更新情報を提供します。ただし、このサービスを管理するフロントエンド

チームは、別のバックエンドチームによって管理されている注文システムデータに直接アクセスすることはできません。バックエンドチームの注文システムはブラックボックスとも呼ばれており、データの構造化に関する情報を収集するのは困難です。しかし、バックエンドチームは、EventBridge によって管理されるイベントバスを通じて、注文データを発行したシステムをセットアップしました。イベントバスからのデータにアクセスしてフロントエンドにルーティングするために、フロントエンドチームは GraphQL API をポイントする新しいターゲットを作成しました AWS AppSync。また、注文の更新情報に関連するデータのみを送信するルールも作成しました。更新が行われると、イベントバスからのデータが GraphQL API に送信されます。API のスキーマによってデータが処理され、フロントエンドに渡されます。

none データソース

データソースを使用する予定がない場合は、データソースを none に設定します。none データソースは、設定後も明示的にデータソースとして分類されますが、ストレージメディアではありません。通常、リゾルバーはある時点で 1 つ以上のデータソースを呼び出してリクエストを処理します。ただし、データソースを操作する必要がない場合もあります。データソースを none に設定すると、リクエストが実行され、データ呼び出しステップはスキップされて、レスポンスが実行されます。

EventBridge セクションの同じユースケースを見てみましょう。スキーマでは、ミューテーションによりステータスの更新が処理されて、サブスクライバーに送信されます。リゾルバーの仕組み上、通常は少なくとも 1 つのデータソース呼び出しがあります。ただし、このシナリオのデータは既にイベントバスによって自動的に送信されています。つまり、ミューテーションによりデータソース呼び出しが実行される必要はなく、注文状況はローカルで処理するだけで済みます。ミューテーションは none に設定され、パススルー値として機能し、データソースは呼び出されません。その後、スキーマにデータが入力され、サブスクライバーに送信されます。

OpenSearch

Amazon OpenSearch Service は、全文検索、データ視覚化、ロギングを実装するためのツールスイートです。このサービスを使用すると、アップロードした構造化データをクエリできます。

このサービスでは、OpenSearch のインスタンスを作成します。これらはノードと呼ばれます。ノードには、少なくとも 1 つのインデックスを追加します。概念的には、インデックスはリレーショナルデータベースのテーブルに少し似ています (ただし、OpenSearch は ACID に準拠していないため、同じようには使用できません)。OpenSearch サービスにアップロードするデータをインデックスに入力します。データがアップロードされると、インデックス内に存在する 1 つ以上のシャードにインデックスが作成されます。シャードは、データの一部を含むインデックスのパーティションのようなもので、他のシャードと別にクエリできます。アップロードされたデータは、ドキュメントと

呼ばれる JSON ファイルとして構造化されます。その後、ノードに対してドキュメント内のデータをクエリできます。

HTTP エンドポイント

HTTP エンドポイントをデータソースとして使用できます。 AWS AppSync は、パラメータやペイロードなどの関連情報を含むリクエストをエンドポイントに送信できます。HTTP レスポンスはリゾルバーに公開され、リゾルバーはオペレーション終了後に最終レスポンスを返します。

データソースを追加する

データソースを作成した場合は、 AWS AppSync サービス、具体的には API にリンクできます。

Console

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   a. ダッシュボードで API を選択します。
   b. サイドバー で [データソース] を選択します。
2. [データソースを作成] を選択します。
   a. データソースに名前を付けます。説明を付けることもできますが、これは任意です。
   b. [データソースタイプ] を選択します。
   c. DynamoDB の場合は、リージョンを選択してから、リージョンのテーブルを選択する必要があります。新しい汎用テーブルロールを作成するか、テーブルの既存のロールをインポートするかを選択することで、テーブルとのインタラクションルールを設定できます。バージョニングを有効にすると、複数のクライアントが同時にデータの更新を試行したとき、リクエストごとにデータのバージョンが自動的に作成されます。バージョニングは、競合の検出と解決を目的に、データの複数のバリアントを保持および管理するために使用されます。また、スキーマ自動生成を有効にすると、データソースが取得され、スキーマ内でそのデータソースにアクセスするのに必要な CRUD、`List`、`Query` オペレーションがいくつか生成されます。

      OpenSearch では、リージョンを選択してから、リージョンのドメイン (クラスター) を選択する必要があります。新しい汎用テーブルロールを作成するか、テーブルの既存のロールをインポートするかを選択することで、ドメインとのインタラクションルールを設定できます。

Lambda の場合は、リージョンを選択してから、そのリージョンの Lambda 関数の ARN を選択する必要があります。新しい汎用テーブルロールを作成するか、テーブルの既存のロールをインポートするかを選択することで、Lambda 関数とのインタラクションルールを設定できます。

HTTP の場合は、HTTP エンドポイントを入力する必要があります。

EventBridge の場合は、リージョンを選択してから、地域のイベントバスを選択する必要があります。新しい汎用テーブルロールを作成するか、テーブルの既存のロールをインポートするかを選択することで、イベントバスとのインタラクションルールを設定できます。

RDS の場合は、リージョンを選択してから、次にシークレットストア (ユーザー名とパスワード)、データベース名、スキーマを選択する必要があります。

None の場合は、実際のデータソースがないデータソースを追加します。これは、リゾルバーを実際のデータソースではなくローカルで処理するためのものです。

> **Note**
>
> 既存のロールをインポートする場合は、信頼ポリシーが必要です。信頼ポリシーの詳細については、「IAM 信頼ポリシー」を参照してください。

3. [作成] を選択します。

> **Note**
>
> または、DynamoDB データソースを作成する場合は、コンソールの [スキーマ] ページに移動し、ページ上部の [リソースの作成] を選択してから、定義済みのモデルを入力してテーブルに変換することもできます。このオプションでは、基本型を入力またはインポートし、パーティションキーを含む基本的なテーブルデータを設定して、スキーマの変更を確認します。

CLI

- `create-data-source` コマンドを実行してデータソースを作成します。

  この特定のコマンドでは、いくつかのパラメータを入力する必要があります。

1. API の api-id です。
2. テーブルの name。
3. データソースの type。選択したデータソースタイプに応じて、service-role-arn と -config タグの入力が必要になる場合があります。

コマンドの例は、次のようになります。

```
aws appsync create-data-source --api-id abcdefghijklmnopqrstuvwxyz
--name data_source_name --type data_source_type --service-role-arn
arn:aws:iam::107289374856:role/role_name --[data_source_type]-config {params}
```

CDK

> **Tip**
>
> CDK を使用する前に、CDK の公式ドキュメントと CDK AWS AppSyncリファレンスを確認することをお勧めします。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html
> 以下の手順では、特定のリソースを追加するために使用するスニペットの一般的な例のみを示しています。これは本番稼働用コードで機能するソリューションとなることを意図したものではありません。また、動作するアプリが既にあることを前提としています。

特定のデータソースを追加するには、コンストラクトをスタックファイルに追加する必要があります。データソースタイプのリストは次のとおりです。

- DynamoDbDataSource
- EventBridgeDataSource
- HttpDataSource
- LambdaDataSource
- NoneDataSource
- OpenSearchDataSource
- RdsDataSource

1. 一般的には、使用しているサービスにインポートディレクティブを追加しなければならない場合があります。たとえば、次の形式に従います。

```
import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'
import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
```

たとえば、 AWS AppSync および DynamoDB サービスをインポートする方法は次のとおりです。

```
import * as appsync from 'aws-cdk-lib/aws-appsync';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
```

2. RDS などの一部のサービスでは、データソースを作成する前にスタックファイルに追加の設定が必要です (VPC の作成、ロール、アクセス認証情報など)。詳細については、関連する CDK ページの例を参照してください。
3. ほとんどのデータソース、特に AWS サービスでは、スタックファイルにデータソースの新しいインスタンスを作成します。通常、結果は以下のようになります。

```
const add_data_source_func = new service_scope.resource_name(scope: Construct,
 id: string, props: data_source_props);
```

その例として、Amazon DynamoDB テーブルの例を次に示します。

```
const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {
  partitionKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
  sortKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
  tableClass: dynamodb.TableClass.STANDARD,
});
```

> **Note**
>
> ほとんどのデータソースには、少なくとも 1 つの必須の props が含まれます (? 記号を使用しないで表記します)。必要とされる props については、CDK ドキュメントを参照してください。

4. 次に、データソースを GraphQL API にリンクする必要があります。おすすめの方法は、パイプラインリゾルバーの関数を作成するときに追加することです。たとえば、以下のスニペットは DynamoDB テーブルのすべての要素をスキャンする関数です。

```
const add_func = new appsync.AppsyncFunction(this, 'func_ID', {
  name: 'func_name_in_console',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('data_source_name_in_console',
 add_ddb_table),
  code: appsync.Code.fromInline(`
      export function request(ctx) {
        return { operation: 'Scan' };
      }

      export function response(ctx) {
        return ctx.result.items;
      }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});
```

dataSource props では、GraphQL API (add_api) を呼び出し、その組み込みメソッドの 1 つ (addDynamoDbDataSource) を使用して、テーブルと GraphQL API を関連付けることができます。引数は、 AWS AppSync コンソール (data_source_name_in_consoleこの例では) とテーブルメソッド () に存在するこのリンクの名前ですadd_ddb_table。このトピックの詳細は、リゾルバーの作成を開始する次のセクションで明らかになります。

データソースをリンクする方法は他にもあります。技術的には、テーブル関数の props リストに api を追加することもできます。たとえば、以下はステップ 3 のスニペットですが、api props には GraphQL API が含まれています。

```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {
  ...
```

```
});

const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {

 ...

  api: add_api
});
```

または、GraphqlApi コンストラクトを個別に呼び出すこともできます。

```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {
  ...
});

const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {
  ...
});

const link_data_source =
 add_api.addDynamoDbDataSource('data_source_name_in_console', add_ddb_table);
```

アソシエーションは関数の props でのみ作成することをおすすめします。それ以外の場合は、 AWS AppSync コンソールでリゾルバー関数をデータソースに手動でリンクするか (コンソール値 を使用し続ける場合data_source_name_in_console)、 のような別の名前で関数に別の関連付けを作成する必要がありますdata_source_name_in_console_2。これは、props による情報処理の方法に制限があるためです。

> **Note**
>
> 変更を確認するには、アプリを再デプロイする必要があります。

IAM 信頼ポリシー

データソースに既存の IAM ロールを使用している場合は、Amazon DynamoDB テーブルなど、AWS リソースでオペレーションを実行するための適切なアクセス許可をそのロールPutItemに付与する必要があります。また、次のポリシー例に示すように、 AWS AppSync がリソースアクセスに使用するには、そのロールの信頼ポリシーを変更する必要があります。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Effect": "Allow",
        "Principal": {
            "Service": "appsync.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
        }
    ]
}
```

また、信頼ポリシーに条件を追加して、必要に応じてデータソースへのアクセスを制限することもできます。現在、SourceArnおよびSourceAccountキーはこれらの条件で使用できます。たとえば、次のポリシーでは、データソースへのアクセスを123456789012 アカウントに制限しています。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "123456789012"
        }
      }
    }
  ]
}
```

または、データソースへのアクセスを、次のポリシーを使用するabcdefghijklmnopqのような特定の API に制限することもできます。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnEquals": {
          "aws:SourceArn": "arn:aws:appsync:us-west-2:123456789012:apis/
abcdefghijklmnopq"
        }
      }
    }
  ]
}
```

以下のポリシーus-east-1を使用して、 などの特定のリージョンからのすべての AWS AppSync APIs へのアクセスを制限できます。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnEquals": {
```

```
            "aws:SourceArn": "arn:aws:appsync:us-east-1:123456789012:apis/*"
          }
        }
      }
    ]
}
```

次のセクション (「リゾルバーの設定」) では、リゾルバーのビジネスロジックを追加し、スキーマのフィールドにアタッチして、データソースのデータを処理します。

詳細については、「IAM ユーザーガイド」の「ロールの修正」を参照してください。

AppSync の AWS Lambda リゾルバーのクロスアカウントアクセスの詳細については、AWS AppSync のクロスアカウント AWS Lambda リゾルバーの構築」を参照してください。 AWS AppSync

でのリゾルバーの設定 AWS AppSync

前のセクションでは、GraphQL スキーマとデータソースを作成し、 AWS AppSync サービスでリンクする方法を学習しました。スキーマでは、クエリとミューテーションに 1 つ以上のフィールド (オペレーション) を設定している場合があります。スキーマには、操作がデータソースに要求するデータの種類が記述されていましたが、それらの操作がデータに対してどのように動作するかは実装されていませんでした。

操作の動作は常にリゾルバーに実装され、リゾルバーは操作を実行するフィールドにリンクされます。リゾルバーの詳細な仕組みについては、リゾルバーページを参照してください。

では AWS AppSync、リゾルバーは、リゾルバーが実行される環境であるランタイムに関連付けられます。ランタイムは、リゾルバーを記述する言語を決定します。現在、APPSYNC_JS (JavaScript) と Apache Velocity (VTL) の 2 つのランタイムがサポートされています。

リゾルバーを実装する場合、リゾルバーは次のような一般的な構造になっています。

- before Step: クライアントからリクエストが送信されると、使用中のスキーマフィールド (通常はクエリ、ミューテーション、サブスクリプション) のリゾルバーにリクエストデータが渡されます。リゾルバーは before step ハンドラーを使用してリクエストデータの処理を開始します。これにより、データがリゾルバーを通過する前に一部の前処理操作を実行できます。
- 関数: before ステップが実行されると、リクエストは関数リストに渡されます。リストの最初の関数がデータソースに対して実行されます。関数は、独自のリクエストハンドラーとレスポンスハン

ドラーを含むリゾルバーのコードのサブセットです。リクエストハンドラーはリクエストデータを取得し、データソースに対して操作を実行します。レスポンスハンドラーは、データソースのレスポンスを処理してからリストに戻します。関数が複数ある場合、リクエストデータはリスト内の次に実行される関数に送信されます。リスト内の関数は、開発者が定義した順序で連続して実行されます。すべての関数が実行されると、最終結果は後のステップに渡されます。

- after step: after step は、GraphQL レスポンスに渡す前に、最終関数のレスポンスに対していくつかの最終オペレーションを実行できるハンドラー関数です。

このフローはパイプラインリゾルバーの例です。パイプラインリゾルバーはどちらのランタイムでもサポートされています。ただし、これはパイプラインリゾルバーで何ができるかを簡単に説明したものです。また、ここでは考えられるリゾルバー構成を 1 つだけ説明しています。サポートされているリゾルバー構成の詳細については、APPSYNC_JS の「JavaScript リゾルバーの概要」または VTL の「リゾルバーマッピングテンプレートの概要」を参照してください。

ご覧のとおり、リゾルバーはモジュール式です。リゾルバーのコンポーネントが正しく動作するためには、他のコンポーネントからの実行状態を覗き見できる必要があります。リゾルバーセクションを見れば、リゾルバーの各コンポーネントには、実行状態に関する重要な情報を一連の引数 (`args`、`context`など) として渡すことができることがわかります。では AWS AppSync、これは によって厳密に処理されます`context`。これは解決対象のフィールドに関する情報を格納するコンテナです。これには、渡される引数、結果、承認データ、ヘッダーデータなど、あらゆるものが含まれます。コンテキストの詳細については、APPSYNC_JS の「リゾルバーコンテキストオブジェクトリファレンス」または VTL の「リゾルバーマッピングテンプレートコンテキストリファレンス」を参照してください。

このコンテキストは、リゾルバーの実装に使用できる唯一のツールではありません。 は、値の生成、エラー処理、解析、変換など、幅広いユーティリティ AWS AppSync をサポートしています。APPSYNC_JS の場合はこちら、VTL の場合はこちらでユーティリティのリストを確認できます。

以下のセクションでは、GraphQL API でリゾルバーを設定する方法を学習します。

トピック

- 基本的なクエリの作成 (JavaScript)
- 基本クエリの作成 (VTL)

基本的なクエリの作成 (JavaScript)

GraphQL リゾルバーは、タイプのスキーマのフィールドをデータソースに接続します。リゾルバーはリクエストを実行するメカニズムです。

AWS AppSync のリゾルバーは、JavaScript を使用して GraphQL 式をデータソースが使用できる形式に変換します。または、マッピングテンプレートを Apache Velocity Template Language (VTL) で記述すると、GraphQL 表現をデータソースで使用できる形式に変換できます。

このセクションでは、JavaScript を使用してリゾルバーを設定する方法を説明します。「リゾルバーのチュートリアル (JavaScript)」セクションでは、JavaScript を使用してリゾルバーを実装する方法に関する詳細なチュートリアルを提供します。「リゾルバーのリファレンス (JavaScript)」セクションでは、JavaScript リゾルバーと共に使用できるユーティリティオペレーションについて説明します。

前述のチュートリアルを使用する前に、このガイドに従うことをお勧めします。

このセクションでは、リゾルバーを作成してクエリとミューテーションを実行するために設定する方法について説明します。

> **Note**
>
> このガイドでは、スキーマが作成済みで、少なくとも 1 つのクエリまたはミューテーションが含まれていることを前提としています。サブスクリプション (リアルタイムデータ) をお探しの場合は、このガイドを参照してください。

このセクションでは、リゾルバーを設定する一般的な手順と、以下のスキーマを使用する例を紹介します。

```
// schema.graphql file

input CreatePostInput {
  title: String
  date: AWSDateTime
}

type Post {
  id: ID!
  title: String
```

```
  date: AWSDateTime
}

type Mutation {
  createPost(input: CreatePostInput!): Post
}

type Query {
  getPost: [Post]
}
```

基本的なクエリリゾルバーの作成

このセクションでは、基本的なクエリリゾルバーを作成する方法を説明します。

Console

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   a. API ダッシュボードで、GraphQL API を選択します。
   b. サイドバーで [スキーマ] を選択します。
2. スキーマとデータソースの詳細を入力します。詳細については、「スキーマの設計」と「データソースを追加する」の各セクションを参照してください。
3. [スキーマ] エディターの横に [リゾルバー] という名前のウィンドウがあります。このボックスには、[スキーマ] ウィンドウで定義されているタイプとフィールドのリストが含まれています。フィールドにはリゾルバーをアタッチできます。ほとんどの場合、フィールド処理にリゾルバーをアタッチすることになります。このセクションでは、簡単なクエリの設定について説明します。[クエリ] 型で、クエリのフィールドの横にある [アタッチ] を選択します。
4. [リゾルバーをアタッチ] ページの [リゾルバータイプ] で、パイプラインリゾルバーとユニットリゾルバーのどちらかを選択できます。これらのリゾルバータイプの詳細については、「リゾルバー」を参照してください。このガイドでは `pipeline resolvers` を使用します。

   > **Tip**
   > パイプラインリゾルバーを作成すると、データソースがパイプライン関数にアタッチされます。関数は、パイプラインリゾルバー自体を作成した後に作成されます。そのため、このページにはこれを設定するオプションはありません。ユニットリゾルバー

を使用する場合は、データソースがリゾルバーに直接関連付けられるため、このページで設定します。

[リゾルバーランタイム] では、[APPSYNC_JS] を選択して JavaScript ランタイムを有効化します。

5. この API のキャッシュは有効にできます。現時点では、この機能をオフにすることをお勧めします。[作成] を選択します。
6. [リゾルバーを編集] ページには、リゾルバーハンドラーとレスポンス (before および after ステップ) のロジックを実装できるリゾルバーコードというコードエディターがあります。詳細については、「JavaScript リゾルバーの概要」を参照してください。

> **Note**
>
> この例では、リクエストを空白のままにし、context からの最後のデータソース結果を返すようにレスポンスを設定します。
>
> ```
> import {util} from '@aws-appsync/utils';
>
> export function request(ctx) {
>     return {};
> }
>
> export function response(ctx) {
>     return ctx.prev.result;
> }
> ```

このセクションの下には、[関数] というテーブルがあります。[関数] では、複数のリゾルバーで再利用できるコードを実装できます。コードを絶えず書き換えたりコピーしたりする代わりに、ソースコードを関数として保存し、必要なときにいつでもリゾルバーに追加できます。

関数はパイプラインのオペレーションリストの大部分を構成します。リゾルバーで複数の関数を使用する場合、関数の順序を設定すると、関数はその順序で順番に実行されます。これらの関数は、リクエスト関数の実行後、レスポンス関数の開始前に実行されます。

新しい関数を追加するには、[関数] で [関数を追加]、[新しい関数の作成] の順に選択します。代わりに [関数の作成] ボタンが表示されて選択できる場合もあります。

a. データソースを選択します。これがリゾルバーの処理対象となるデータソースになります。

> **Note**
>
> この例では、id で Post オブジェクトを取得する getPost にリゾルバーをアタッチします。このスキーマ用の DynamoDB テーブルをすでにセットアップ済みであると仮定します。そのパーティションキーは id に設定されており、空です。

b. Function name を入力します。

c. [関数コード] で、関数の動作を実装する必要があります。わかりにくいかもしれませんが、各関数には独自のローカルのリクエストハンドラーとレスポンスハンドラーがあります。リクエストが実行され、次にデータソース呼び出しが実行されてリクエストが処理され、データソースのレスポンスがレスポンスハンドラーによって処理されます。結果は context オブジェクトに保存されます。その後、リスト内の次の関数が実行されるか、それが最後の関数であればステップ後のレスポンスハンドラーに渡されます。

> **Note**
>
> この例では、データソースから Post オブジェクトのリストを取得する getPost にリゾルバーをアタッチします。リクエスト関数はテーブルにデータをリクエストし、テーブルはそのレスポンスをコンテキスト (ctx) に渡し、レスポンスはコンテキストで結果を返します。 AWS AppSyncの強度は、他の AWS サービスとの相互接続性にあります。DynamoDB を使用しているため、このような作業を簡略化するための一連のオペレーションがあります。他のデータソースタイプについても共通する例がいくつかあります。
> コードは以下のようになります。

```
import { util } from '@aws-appsync/utils';

/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
```

```
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

このステップでは、次の 2 つの関数を追加しました。

- `request`: リクエストハンドラーはデータソースに対して取得オペレーションを実行します。引数には、コンテキストオブジェクト (`ctx`)、または特定のオペレーションを実行するすべてのリゾルバーが利用できるデータが含まれます。たとえば、認可データや解決対象のフィールド名などが含まれる場合があります。return ステートメントは `Scan` オペレーションを実行します (例はこちらを参照)。DynamoDB を使用しているため、そのサービスの一部のオペレーションを使用できます。このスキャンでは、テーブル内のすべての項目の基本的なフェッチが実行されます。このオペレーションの結果は、レスポンスハンドラーに渡される前に `result` コンテナーとしてコンテキストオブジェクトに保存されます。`request` はパイプライン内のレスポンス前に実行されます。
- `response`: `request` の出力を返すレスポンスハンドラー。引数は、更新されたコンテキストオブジェクトで、return ステートメントは `ctx.prev.result` です。ガイドのこの段階では、この値についてはよくわからないかもしれません。`ctx` は context object オブジェクトを参照します。`prev` はパイプライン内の直前のオペレーション、この例では `request` を参照します。`result` には、リゾルバーがパイプラインを経由して移動する際の結果が含まれます。すべてをまとめると、`ctx.prev.result` は最後に実行されたオペレーションの結果であるリクエストハンドラーを返します。

d. 完了したら、[作成] を選択します。

7. リゾルバー画面に戻り、[関数] で [関数を追加] ドロップダウンを選択して、関数を関数リストに追加します。
8. [保存] を選択してリゾルバーを更新します。

CLI

関数を追加するには

- create-function コマンドを使用してパイプラインリゾルバー用の関数を作成します。

  この特定のコマンドでは、いくつかのパラメータを入力する必要があります。

  1. API の api-id です。
  2. AWS AppSync コンソールの 関数nameの 。
  3. data-source-name、すなわち関数で使用されるデータソースの名前。すでに作成され、 AWS AppSync サービス内の GraphQL API にリンクされている必要があります。
  4. runtime、すなわち関数の環境と言語。JavaScript の場合、name は APPSYNC_JS で、runtime は 1.0.0 である必要があります。
  5. code、すなわち関数のリクエストハンドラーとレスポンスハンドラー。手動でも入力できますが、.txt ファイル (または同様の形式) に追加して引数として渡すほうがはるかに簡単です。

  > **Note**
  >
  > クエリコードは引数として渡されるファイルにあります。

```
import { util } from '@aws-appsync/utils';

/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

コマンドの例は、次のようになります。

```
aws appsync create-function \
--api-id abcdefghijklmnopqrstuvwxyz \
--name get_posts_func_1 \
--data-source-name table-for-posts \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file://~/path/to/file/{filename}.{fileType}
```

出力は CLI に返されます。例を示します。

```
{
    "functionConfiguration": {
        "functionId": "ejglgvmcabdn7lx75ref4qeig4",
        "functionArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/functions/ejglgvmcabdn7lx75ref4qeig4",
        "name": "get_posts_func_1",
        "dataSourceName": "table-for-posts",
        "maxBatchSize": 0,
        "runtime": {
            "name": "APPSYNC_JS",
            "runtimeVersion": "1.0.0"
        },
        "code": "Code output goes here"
    }
}
```

> **Note**
>
> functionId は、必ずどこかに記録しておいてください。これは、リゾルバーに関数をアタッチする際に使用します。

リゾルバーを作成するには

- create-resolver コマンドを実行して Query のパイプライン関数を作成します。

  この特定のコマンドでは、いくつかのパラメータを入力する必要があります。

1. API の `api-id` です。
2. `type-name`、すなわちスキーマ内の特別なオブジェクトタイプ (クエリ、ミューテーション、サブスクリプション)。
3. `field-name`、すなわちリゾルバーをアタッチする特別なオブジェクトタイプ内のフィールドオペレーション。
4. `kind`。ユニットまたはパイプラインリゾルバーを指定します。これを `PIPELINE` に設定すると、パイプライン関数が有効になります。
5. `pipeline-config`、すなわちリゾルバーにアタッチする関数。関数の `functionId` 値がわかっていることを確認してください。リストの順序は重要です。
6. `runtime`。以前は `APPSYNC_JS` (JavaScript) でした。`runtimeVersion` は現在、`1.0.0` です。
7. `code`。before および after ステップハンドラーが含まれています。

> **Note**
>
> クエリコードは引数として渡されるファイルにあります。

```
import { util } from '@aws-appsync/utils';

/**
 * Sends a request to `put` an item in the DynamoDB data source
 */
export function request(ctx) {
  const { id, ...values } = ctx.args;
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}

/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}
```

コマンドの例は、次のようになります。

```
aws appsync create-resolver \
--api-id abcdefghijklmnopqrstuvwxyz \
--type-name Query \
--field-name getPost \
--kind PIPELINE \
--pipeline-config functions=ejglgvmcabdn7lx75ref4qeig4 \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}
```

出力は CLI に返されます。例を示します。

```
{
    "resolver": {
        "typeName": "Mutation",
        "fieldName": "getPost",
        "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Mutation/resolvers/getPost",
        "kind": "PIPELINE",
        "pipelineConfig": {
            "functions": [
                "ejglgvmcabdn7lx75ref4qeig4"
            ]
        },
        "maxBatchSize": 0,
        "runtime": {
            "name": "APPSYNC_JS",
            "runtimeVersion": "1.0.0"
        },
        "code": "Code output goes here"
    }
}
```

CDK

> **Tip**
>
> CDK を使用する前に、CDK の公式ドキュメントと CDK AWS AppSyncリファレンスを確認することをお勧めします。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html
> 以下の手順では、特定のリソースを追加するために使用するスニペットの一般的な例のみを示しています。これは本番稼働用コードで機能するソリューションとなることを意図したものではありません。また、動作するアプリが既にあることを前提としています。

基本的なアプリには以下のものが必要です。

1. サービス import ディレクティブ
2. スキーマのコード
3. データソースジェネレーター
4. 関数コード
5. リゾルバーのコード

「スキーマの設計」と「データソースを追加する」の各セクションによれば、スタックファイルには以下の形式の import ディレクティブが含まれます。

```
import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'
import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
```

> **Note**
>
> 前のセクションでは、 AWS AppSync コンストラクトをインポートする方法のみについて説明しました。実際のコードでは、アプリを実行するためだけにより多くのサービスをインポートする必要があります。この例では、非常にシンプルな CDK アプリを作成する場合、少なくとも DynamoDB テーブルであるデータソースとともに AWS AppSync サービスをインポートします。また、アプリをデプロイするには、いくつかの追加コンストラクトをインポートする必要があります。
>
> ```
> import * as cdk from 'aws-cdk-lib';
> import * as appsync from 'aws-cdk-lib/aws-appsync';
> ```

```
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
import { Construct } from 'constructs';
```

それぞれについて概説すると、次のようになります。

- `import * as cdk from 'aws-cdk-lib';`: これにより、CDK アプリとスタックなどのコンストラクトを定義できます。また、メタデータの操作など、アプリケーションに役立つユーティリティ関数もいくつか含まれています。このインポートディレクティブには精通しているものの、cdk コアライブラリがここで使用される理由が不明の場合は、「マイグレーション」ページを参照してください。
- `import * as appsync from 'aws-cdk-lib/aws-appsync';`: これにより AWS AppSync サービスがインポートされます。
- `import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';`: これにより DynamoDB サービスがインポートされます。
- `import { Construct } from 'constructs';`: ルート コンストラクトを定義するにはこれが必要です。

インポートの種類は、呼び出すサービスによって異なります。例については、CDK のドキュメントを参照することをお勧めします。ページ上部のスキーマは、CDK アプリでは `.graphql` ファイルとして個別のファイルになります。スタックファイルでは、次の形式でスキーマを新しい GraphQL に関連付けることができます。

```
const add_api = new appsync.GraphqlApi(this, 'graphQL-example', {
  name: 'my-first-api',
  schema: appsync.SchemaFile.fromAsset(path.join(__dirname, 'schema.graphql')),
});
```

Note

スコープ `add_api` では、`new` キーワードの後に `appsync.GraphqlApi(scope: Construct, id: string , props: GraphqlApiProps)` を続けて新しい GraphQL API が追加されています。スコープは `this` で、CFN ID が `graphQL-example` で、props は `my-first-api` (コンソールに表示される API の名前) および `schema.graphql` (スキーマファイルへの絶対パス) です。

データソースを追加するには、まずデータソースをスタックに追加する必要があります。次に、ソース固有のメソッドを使用して GraphQL API に関連付ける必要があります。この関連付けは、リゾルバー関数を作成したときに行われます。その間に、dynamodb.Table を使用して DynamoDB テーブルを作成する例を見てみましょう。

```
const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
  partitionKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
});
```

> **Note**
>
> この例でこれを使用すると、CFN ID が posts-table で、パーティションキーが id (S) という新しい DynamoDB テーブルが追加されます。

次に、スタックファイルにリゾルバーを実装する必要があります。DynamoDB テーブル内のすべての項目をスキャンする簡単なクエリの例を以下に示します。

```
const add_func = new appsync.AppsyncFunction(this, 'func-get-posts', {
  name: 'get_posts_func_1',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('table-for-posts', add_ddb_table),
  code: appsync.Code.fromInline(`
      export function request(ctx) {
        return { operation: 'Scan' };
      }

      export function response(ctx) {
        return ctx.result.items;
      }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});

new appsync.Resolver(this, 'pipeline-resolver-get-posts', {
  add_api,
  typeName: 'Query',
  fieldName: 'getPost',
```

```
  code: appsync.Code.fromInline(`
      export function request(ctx) {
        return {};
      }

      export function response(ctx) {
        return ctx.prev.result;
      }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
  pipelineConfig: [add_func],
});
```

Note

最初に、add_func という関数を作成しました。この作成順序は少し直観に反するような印象を持つかもしれませんが、リゾルバー自体を作成する前に、パイプラインリゾルバーで関数を作成する必要があります。関数は次の形式に従います。

```
AppsyncFunction(scope: Construct, id: string, props: AppsyncFunctionProps)
```

スコープは thisで、CFN ID が func-get-posts で、props には実際に関数の詳細が含まれています。props 内には以下が含まれています。

- AWS AppSync コンソールに存在する関数nameの (get_posts_func_1) 。
- 以前に作成した GraphQL API (add_api)。
- データソース。これは、データソースを GraphQL API 値にリンクし、それを関数にアタッチするポイントです。作成したテーブル (add_ddb_table) を取得し、GraphqlApi メソッド (addDynamoDbDataSource) のいずれかを使用して GraphQL API (add_api) にアタッチします。ID 値 (table-for-posts) は AWS AppSync コンソールに表示されるデータソースの名前です。ソース固有のメソッドの一覧については、次のページを参照してください。
  - DynamoDbDataSource
  - EventBridgeDataSource
  - HttpDataSource
  - LambdaDataSource
  - NoneDataSource

- OpenSearchDataSource
- RdsDataSource

- コードには関数のリクエストハンドラーとレスポンスハンドラーが含まれており、シンプルなスキャンを実行し、結果を返します。
- ランタイムでは、APPSYNC_JS ランタイムバージョン 1.0.0 を使用することを指定しています。現在 APPSYNC_JS で使用できるのはこのバージョンだけであることに注意してください。

次に、関数をパイプラインリゾルバーにアタッチする必要があります。リゾルバーは次の形式で作成しています。

```
Resolver(scope: Construct, id: string, props: ResolverProps)
```

スコープは `this`で、CFN ID が `pipeline-resolver-get-posts` で、props には実際に関数の詳細が含まれています。props 内には以下が含まれています。

- 以前に作成した GraphQL API (`add_api`)。
- 特別なオブジェクトタイプ名。これはクエリオペレーションであり、`Query` 値を追加しただけです。
- フィールド名 (`getPost`) は、`Query` タイプのスキーマ内にあるフィールドの名前です。
- コードには before ハンドラーと after ハンドラーが含まれています。この例では、関数がオペレーションを実行した後の context 内の結果を返すだけです。
- ランタイムでは、APPSYNC_JS ランタイムバージョン 1.0.0 を使用することを指定しています。現在 APPSYNC_JS で使用できるのはこのバージョンだけであることに注意してください。
- パイプライン config には、作成した関数 (`add_func`) への参照が含まれています。

この例で何が起こったかを要約すると、リクエストとレスポンスハンドラーを実装した AWS AppSync 関数が表示されました。この関数はデータソースとやりとりする役割を持っています。リクエストハンドラーは にScanオペレーションを送信し AWS AppSync、DynamoDB データソースに対して実行するオペレーションを指示します。レスポンスハンドラーは項目のリスト (`ctx.result.items`) を返します。その後、項目のリストは `Post` GraphQL タイプに自動的にマッピングされました。

基本的なミューテーションリゾルバーの作成

このセクションでは、基本的なミューテーションリゾルバーを作成する方法を説明します。

Console

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   a. API ダッシュボードで、GraphQL API を選択します。
   b. サイドバーで [スキーマ] を選択します。
2. [リゾルバー] セクションと[ミューテーションタイプ] で、フィールドの横にある [アタッチ] を選択します。

   > **Note**
   > この例では、`createPost` のリゾルバーをアタッチすることで、`Post` オブジェクトをテーブルに追加します。前のセクションと同じ DynamoDB テーブルを使用していると仮定します。そのパーティションキーは `id` に設定されており、空です。

3. [リゾルバーをアタッチ] ページの [リゾルバータイプ] で、[`pipeline resolvers`] を選択します。リゾルバーの詳細を確認する場合は、こちらを参照してください。[リゾルバーランタイム] では、[APPSYNC_JS] を選択して JavaScript ランタイムを有効化します。
4. この API のキャッシュは有効にできます。現時点では、この機能をオフにすることをお勧めします。[作成] を選択します。
5. [関数の追加] を選択して、[新しい関数を作成] を選択します。代わりに [関数の作成] ボタンが表示されて選択できる場合もあります。
   a. データソースを選択します。これは、ミューテーション時にデータを操作する際のソースになります。
   b. `Function name` を入力します。
   c. [関数コード] で、関数の動作を実装する必要があります。これはミューテーションであるため、リクエストにより、呼び出されたデータソースに対してある程度状態が変化するオペレーションが実行されると理想的です。結果はレスポンス関数によって処理されます。

Note

createPost はパラメータがデータとして含まれているテーブルに新しい Post ものを追加します (入れます)。次のように追加します。

```
import { util } from '@aws-appsync/utils';

/**
 * Sends a request to `put` an item in the DynamoDB data source
 */
export function request(ctx) {
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({id: util.autoId()}),
    attributeValues: util.dynamodb.toMapValues(ctx.args.input),
  };
}

/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}
```

このステップでは、request と response の 2 つの関数を追加しました。

- request: リクエストハンドラーはコンテキストを引数として受け入れます。リクエストハンドラーの return ステートメントは、組み込みの DynamoDB オペレーションである PutItem コマンドを実行します (例については、こちらまたはこちらを参照してください)。PutItem このコマンドは、パーティション key 値 (util.autoid() によって自動的に生成) とコンテキストの引数の入力による attributes (リクエストで渡される値) を取得して、Post オブジェクトを DynamoDB テーブルに追加します。key は、id で、attributes は date および title フィールド引数です。どちらも util.dynamodb.toMapValuesヘルパーを経由して実行され、DynamoDB テーブルを使用します。

- `response`: レスポンスは、更新されたコンテキストを受け入れ、リクエストハンドラーの結果を返します。

d. 完了したら、[作成] を選択します。

6. リゾルバー画面に戻り、[関数] で [関数を追加] ドロップダウンを選択して、関数を関数リストに追加します。
7. [保存] を選択してリゾルバーを更新します。

CLI

関数を追加するには

- `create-function` コマンドを使用してパイプラインリゾルバー用の関数を作成します。

  この特定のコマンドでは、いくつかのパラメータを入力する必要があります。

  1. API の `api-id` です。
  2. AWS AppSync コンソールの 関数nameの 。
  3. `data-source-name`、すなわち関数で使用されるデータソースの名前。すでに作成され、 AWS AppSync サービス内の GraphQL API にリンクされている必要があります。
  4. `runtime`、すなわち関数の環境と言語。JavaScript の場合、name は `APPSYNC_JS` で、runtime は `1.0.0` である必要があります。
  5. `code`、すなわち関数のリクエストハンドラーとレスポンスハンドラー。手動でも入力できますが、.txt ファイル (または同様の形式) に追加して引数として渡すほうがはるかに簡単です。

  > **Note**
  >
  > クエリコードは引数として渡されるファイルにあります。

```
import { util } from '@aws-appsync/utils';

/**
 * Sends a request to `put` an item in the DynamoDB data source
 */
export function request(ctx) {
  return {
    operation: 'PutItem',
```

```
    key: util.dynamodb.toMapValues({id: util.autoId()}),
    attributeValues: util.dynamodb.toMapValues(ctx.args.input),
  };
}

/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}
```

コマンドの例は、次のようになります。

```
aws appsync create-function \
--api-id abcdefghijklmnopqrstuvwxyz \
--name add_posts_func_1 \
--data-source-name table-for-posts \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}
```

出力は CLI に返されます。例を示します。

```
{
    "functionConfiguration": {
        "functionId": "vulcmbfcxffiram63psb4dduoa",
        "functionArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/functions/vulcmbfcxffiram63psb4dduoa",
        "name": "add_posts_func_1",
        "dataSourceName": "table-for-posts",
        "maxBatchSize": 0,
        "runtime": {
            "name": "APPSYNC_JS",
            "runtimeVersion": "1.0.0"
        },
        "code": "Code output foes here"
    }
}
```

> **Note**
>
> functionId は、必ずどこかに記録しておいてください。これは、リゾルバーに関数をアタッチする際に使用します。

リゾルバーを作成するには

- create-resolver コマンドを実行して Mutation のパイプライン関数を作成します。

  この特定のコマンドでは、いくつかのパラメータを入力する必要があります。

  1. API の api-id です。
  2. type-name、すなわちスキーマ内の特別なオブジェクトタイプ (クエリ、ミューテーション、サブスクリプション)。
  3. field-name、すなわちリゾルバーをアタッチする特別なオブジェクトタイプ内のフィールドオペレーション。
  4. kind。ユニットまたはパイプラインリゾルバーを指定します。これを PIPELINE に設定すると、パイプライン関数が有効になります。
  5. pipeline-config、すなわちリゾルバーにアタッチする関数。関数の functionId 値がわかっていることを確認してください。リストの順序は重要です。
  6. runtime。以前は APPSYNC_JS (JavaScript) でした。runtimeVersion は現在、1.0.0 です。
  7. code。before および after ステップが含まれています。

  > **Note**
  >
  > クエリコードは引数として渡されるファイルにあります。

```
import { util } from '@aws-appsync/utils';

/**
 * Sends a request to `put` an item in the DynamoDB data source
 */
export function request(ctx) {
  const { id, ...values } = ctx.args;
  return {
```

```
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}

/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}
```

コマンドの例は、次のようになります。

```
aws appsync create-resolver \
--api-id abcdefghijklmnopqrstuvwxyz \
--type-name Mutation \
--field-name createPost \
--kind PIPELINE \
--pipeline-config functions=vulcmbfcxffiram63psb4dduoa \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}
```

出力は CLI に返されます。例を示します。

```
{
    "resolver": {
        "typeName": "Mutation",
        "fieldName": "createPost",
        "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Mutation/resolvers/createPost",
        "kind": "PIPELINE",
        "pipelineConfig": {
            "functions": [
                "vulcmbfcxffiram63psb4dduoa"
            ]
        },
        "maxBatchSize": 0,
        "runtime": {
            "name": "APPSYNC_JS",
```

```
            "runtimeVersion": "1.0.0"
        },
        "code": "Code output goes here"
    }
}
```

CDK

> **Tip**
>
> CDK を使用する前に、CDK の公式ドキュメントと CDK AWS AppSyncリファレンスを確認することをお勧めします。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html
> 以下の手順では、特定のリソースを追加するために使用するスニペットの一般的な例のみを示しています。これは本番稼働用コードで機能するソリューションとなることを意図したものではありません。また、動作するアプリが既にあることを前提としています。

- 同じプロジェクトに属していると仮定して、ミューテーションを行うには、クエリのようにスタックファイルにミューテーションを追加できます。以下は、テーブルに新しい Post を追加するミューテーションの修正済み関数とリゾルバーです。

```
const add_func_2 = new appsync.AppsyncFunction(this, 'func-add-post', {
  name: 'add_posts_func_1',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('table-for-posts-2', add_ddb_table),
      code: appsync.Code.fromInline(`
          export function request(ctx) {
            return {
              operation: 'PutItem',
              key: util.dynamodb.toMapValues({id: util.autoId()}),
              attributeValues: util.dynamodb.toMapValues(ctx.args.input),
            };
          }

          export function response(ctx) {
            return ctx.result;
          }
      `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
```

```
});

new appsync.Resolver(this, 'pipeline-resolver-create-posts', {
  add_api,
  typeName: 'Mutation',
  fieldName: 'createPost',
      code: appsync.Code.fromInline(`
          export function request(ctx) {
            return {};
          }

          export function response(ctx) {
            return ctx.prev.result;
          }
      `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
  pipelineConfig: [add_func_2],
});
```

> **Note**
>
> このミューテーションとクエリの構造は似ているため、ここではミューテーションを実行するために行った変更について説明します。
> この関数では、テーブルに Posts を追加していることを反映して CFN ID を func-add-post に、名前を add_posts_func_1 に変更しました。データソースでは、 addDynamoDbDataSource メソッドで必要なtable-for-posts-2ため、AWS AppSync コンソールでテーブル (add_ddb_table) に新しい関連付けを行いました。この新しい関連付けは、先ほど作成したのと同じテーブルをまだ使用していますが、 AWS AppSync コンソールには 2 つの接続があります。1 つはクエリ用でtable-for-posts、もう 1 つはミューテーション用ですtable-for-posts-2。コードは、id 値を自動的に生成し、残りのフィールドのクライアントの入力を受け入れることで、Post を追加するように変更されました。
> リゾルバーでは、テーブルに Posts を追加していることを反映して ID 値を pipeline-resolver-create-posts に変更しました。スキーマのミューテーションを反映するため、タイプ名が Mutation に変更され、名前が createPost に変更されました。パイプラインの config は、新しいミューテーション関数 add_func_2 に設定されました。

この例で何が起こっているかを要約すると、 は createPostフィールドで定義された引数を GraphQL スキーマから DynamoDB オペレーション AWS AppSync に自動的に変換します。util.autoId() ヘルパーを使用して自動的に作成される id のキーを使用して、DynamoDB にレコードが保存されます。 AWS AppSync コンソールで行われたリクエストまたはそれ以外のリクエストからコンテキスト引数 (ctx.args.input) に渡す他のすべてのフィールドは、テーブルの属性として保存されます。キーと属性は両方とも、util.dynamodb.toMapValues(values) ヘルパーを使用して互換性のある DynamoDB 形式に自動的にマッピングされます。

AWS AppSync は、リゾルバーを編集するためのテストおよびデバッグワークフローもサポートしています。モック context オブジェクトを使用して、呼び出す前にテンプレートでの変換後の値を確認できます。また、クエリを実行する際にデータソースへのリクエスト全体をインタラクティブに表示することもできます。詳細については、「リゾルバーのテストとデバッグ (JavaScript)」および「モニタリングとロギング」を参照してください。

高度なリゾルバー

「スキーマの設計」のオプションのページネーションセクションに従っている場合でも、ページネーションを利用するにはリゾルバーをリクエストに追加する必要があります。この例では、getPosts と呼ばれるクエリページネーションを使用して、リクエストされた内容の一部だけを一度に返します。このフィールドのリゾルバーのコードは以下のようになります。

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  const { limit = 20, nextToken } = ctx.args;
  return { operation: 'Scan', limit, nextToken };
}

/**
 * @returns the result of the `put` operation
 */
export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

リクエストでは、リクエストのコンテキストを渡します。この例では、limit は *20* で、最初のクエリでは最大 20 Posts を返します。nextToken カーソルはデータソースの最初の Post エントリに固定されています。これらは args に渡されます。その後、リクエストは最初の Post からスキャ

ン制限数までスキャンを実行します。データソースは結果をコンテキストに保存し、その結果はレスポンスに渡されます。レスポンスは取得した Posts を返し、nextToken が制限数直後の Post エントリに設定されます。次のリクエストも送信され、まったく同じ処理が行われますが、最初のクエリの直後のオフセットから開始されます。これらの種類のリクエストは、並列ではなくシーケンシャルに行われることに注意してください。

AWS AppSync でのリゾルバーのテストとデバッグ (JavaScript)

AWS AppSync は、データソースに対して GraphQL フィールドでリゾルバーを実行します。パイプラインリゾルバーを操作する場合、関数はデータソースとやりとりします。「JavaScript リゾルバーの概要」で説明したように、関数は JavaScript で記述され、ランタイムで実行されるリクエストハンドラーとレスポンスハンドラーを使用してデータソースと通信しますAPPSYNC_JS。これにより、ユーザーはカスタムロジック条件を提供でき、データソースと通信する前後にロジックおよび条件を適用できます。

デベロッパーがこれらのリゾルバーを記述、テスト、デバッグできるように、 AWS AppSync コンソールには、モックデータを含む GraphQL リクエストとレスポンスを個々のフィールドリゾルバーに作成するためのツールも用意されています。さらに、 AWS AppSync コンソールでクエリ、ミューテーション、サブスクリプションを実行し、Amazon CloudWatch からのリクエスト全体の詳細なログストリームを表示できます。ログストリームにはデータソースからの結果も含まれています。

モックデータを使用したテスト

GraphQL リゾルバーが呼び出されるときに、そのリゾルバーには、リクエストに関する情報が含まれている context オブジェクトが含まれています。このオブジェクトには、クライアントからの引数、ID 情報、および親 GraphQL フィールドからのデータが含まれています。また、データソースからの結果も含まれていて、それをレスポンスハンドラーで使用できます。この構造およびプログラミング時に使用可能なヘルパーユーティリティの詳細については、「リゾルバーコンテキストオブジェクトリファレンス」を参照してください。

リゾルバー関数を記述または編集する場合に、モックまたはコンテキストのテストの オブジェクトをコンソールエディタに渡すことができます。これにより、実際にデータソースに対して実行することなく、リクエストとレスポンスハンドラーの両方でどのように評価されるかを確認できます。例えば、テストの firstname: Shaggy 引数を渡して、テンプレートのコードで ctx.args.firstname を使用している場合にその引数がどのように評価されるかを確認できます。任意のユーティリティヘルパー (util.autoId()、util.time.nowISO8601() など) での評価をテストすることもできます。

リゾルバーのテスト

この例では、 AWS AppSync コンソールを使用してリゾルバーをテストします。

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   a. API ダッシュボードで、GraphQL API を選択します。
   b. サイドバーで [関数] を選択します。
2. 既存の関数を選択します。
3. [関数の更新] ページの上部で、[テストコンテキストを選択] を選択し、[新しいコンテキストを作成] を選択します。
4. サンプルコンテキストオブジェクトを選択するか、下の [テストコンテキストを設定] ウィンドウで JSON を手動で入力します。
5. テキストコンテキスト名を入力します。
6. [保存] ボタンを選択します。
7. この模擬コンテキストオブジェクトを使用してリゾルバーを評価するには、[Run Test] を選択します。

例えば、オブジェクトに対して自動 ID 生成を使用して Amazon DynamoDB に保存する Dog の GraphQL タイプを保存するアプリケーションがあるとします。また、一部の値を GraphQL ミューテーションの引数から書き込み、レスポンスが特定の 1 人のユーザーにのみ表示されるようにします。次のスニペットがスキーマがどのようになっているかを示します。

```
type Dog {
  breed: String
  color: String
}

type Mutation {
  addDog(firstname: String, age: Int): Dog
}
```

AWS AppSync 関数を記述し、リaddDogゾルバーに追加してミューテーションを処理できます。AWS AppSync 関数をテストするには、次の例のようにコンテキストオブジェクトを入力します。次の例には、クライアントから引数として `name` と `age` があり、`identity` オブジェクトに入力されている `username` があります。

```
{
```

```
    "arguments" : {
        "firstname": "Shaggy",
        "age": 4
    },
    "source" : {},
    "result" : {
        "breed" : "Miniature Schnauzer",
        "color" : "black_grey"
    },
    "identity": {
        "sub" : "uuid",
        "issuer" : " https://cognito-idp.{region}.amazonaws.com/{userPoolId}",
        "username" : "Nadia",
        "claims" : { },
        "sourceIp" :[  "x.x.x.x" ],
        "defaultAuthStrategy" : "ALLOW"
    }
}
```

次のコードを使用して AWS AppSync 関数をテストできます。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id: util.autoId() }),
    attributeValues: util.dynamodb.toMapValues(ctx.args),
  };
}

export function response(ctx) {
  if (ctx.identity.username === 'Nadia') {
    console.log("This request is allowed")
    return ctx.result;
  }
  util.unauthorized();
}
```

評価後のリクエストとレスポンスハンドラーには、テストコンテキストオブジェクトからのデータと、`util.autoId()` から生成された値があります。また、`username` を `Nadia` 以外の値に変更し

た場合は、認可チェックが失敗するため、結果は返されません。きめ細かなアクセス制御の詳細については、「認可のユースケース」を参照してください。

AWS AppSync APIs を使用したリクエストハンドラーとレスポンスハンドラーのテスト

EvaluateCode API コマンドを使用して、モックデータを使用してコードをリモートでテストできます。コマンドを使い始めるには、ポリシーに appsync:evaluateMappingCode アクセス許可を追加していることを確認してください。例:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateCode",
            "Resource": "arn:aws:appsync:<region>:<account>:*"
        }
    ]
}
```

AWS CLI または AWS SDK を使用してコマンドを活用できます。例えば、前のセクションで Dog スキーマとその AWS AppSync 関数のリクエストおよびレスポンスハンドラーを見てみましょう。ローカルステーションの CLI を使用して、コードを code.js という名前のファイルに保存し、context オブジェクトを context.json という名前のファイルに保存します。シェルから次のコマンドを実行します。

```
$ aws appsync evaluate-code \
  --code file://code.js \
  --function response \
  --context file://context.json \
  --runtime name=APPSYNC_JS,runtimeVersion=1.0.0
```

レスポンスには、ハンドラーから返されたペイロードを含む evaluationResult が含まれます。また、評価中にハンドラーによって生成されたログのリストを保持する logs オブジェクトも含まれています。これにより、コード実行のデバッグが容易になり、評価に関する情報を確認してトラブルシューティングに役立てることができます。例:

```
{
    "evaluationResult": "{\"breed\":\"Miniature Schnauzer\",\"color\":\"black_grey\"}",
    "logs": [
        "INFO - code.js:13:5: \"This request is allowed\""
    ]
}
```

evaluationResult は JSON として解析でき、以下を実現します。

```
{
  "breed": "Miniature Schnauzer",
  "color": "black_grey"
}
```

SDK を使用すると、お気に入りのテストスイートのテストを簡単に組み込んで、ハンドラーの動作を検証できます。Jest テストフレームワークを使用してテストを作成することをお勧めしますが、どのテストスイートでも問題ありません。次のスニペットは、仮説検証の実行を示しています。評価レスポンスは有効な JSON であることを想定しているので、JSON.parseを使用して文字列レスポンスから JSON を取得することに注意してください。

```
const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })
const runtime = {name:'APPSYNC_JS',runtimeVersion:'1.0.0')

test('request correctly calls DynamoDB', async () => {
  const code = fs.readFileSync('./code.js', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateCode({ code, context, runtime, function:
 'request' }).promise()
  const result = JSON.parse(response.evaluationResult)

  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})
```

これにより、次のような結果が得られます。

```
Ran all test suites.
```

```
> jest

PASS ./index.test.js
# request correctly calls DynamoDB (543 ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 totalTime: 1.511 s, estimated 2 s
```

ライブクエリのデバッグ

本番アプリケーションをデバッグするためのend-to-endのテストとログ記録に代わるものはありません。 AWS AppSync では、Amazon CloudWatch を使用してエラーとリクエストの詳細全体をログ記録できます。さらに、 AWS AppSync コンソールを使用して、各リクエストの GraphQL クエリ、ミューテーション、サブスクリプション、ライブストリームログデータをテストし、クエリエディタに戻し、リアルタイムでデバッグできます。サブスクリプションに関して表示されるログは接続時の情報です。

これを実行するには、「モニタリングとロギング」で説明しているように、Amazon CloudWatch Logs を事前に有効にしておく必要があります。次に、 AWS AppSync コンソールでクエリタブを選択し、有効な GraphQL クエリを入力します。右下のセクションで、[ログ] ウィンドウをクリックしてドラッグし、ログビューを開きます。ページの上部にある再生矢印アイコンを選択して GraphQL クエリを実行します。しばらくすると、そのオペレーションのリクエストとレスポンスの完全なログが、このセクションにストリーミングされ、コンソールで表示できます。

AWS AppSync でのパイプラインリゾルバーの設定と使用 (JavaScript)

AWS AppSync は GraphQL フィールドでリゾルバーを実行します。場合によっては、アプリケーションは 1 つの GraphQL フィールドを解決するために複数のオペレーションを実行する必要があります。パイプラインリゾルバーを使用すると、開発者は関数と呼ばれるオペレーションを構成して順番に実行できます。パイプラインリゾルバーは、たとえば、フィールドのデータを取得する前に承認チェックを実行する必要があるアプリケーションに便利です。

JavaScript パイプラインリゾルバーのアーキテクチャの詳細については、「JavaScript リゾルバーの概要」を参照してください。

ステップ 1: パイプラインリゾルバーを作成する

AWS AppSync コンソールで、スキーマページに移動します。

以下のスキーマを保存します。

```
schema {
    query: Query
    mutation: Mutation
}

type Mutation {
    signUp(input: Signup): User
}

type Query {
    getUser(id: ID!): User
}

input Signup {
    username: String!
    email: String!
}

type User {
    id: ID!
    username: String
    email: AWSEmail
}
```

パイプラインリゾルバーをミューテーションタイプの SignUp フィールドに書き込みます。右側の [ミューテーション] タイプに入力し、[signUp ミューテーション] フィールドの横の [アタッチ] を選択します。リゾルバーを `pipeline resolver` および `APPSYNC_JS` ランタイムに設定し、リゾルバーを作成します。

パイプラインリゾルバーは、最初に E メールアドレスの入力を検証してからシステムにユーザーを保存することで、ユーザーをサインアップします。E メールの検証を validateEmail 関数内にカプセル化し、ユーザーの保存を saveUser 関数内にカプセル化します。validateEmail 関数が最初に実行され、E メールが有効であれば、saveUser 関数が実行されます。

実行フローは以下のようになります。

1. Mutation.SignUp リゾルバーリクエストハンドラー
2. validateEmail 関数
3. saveUser 関数
4. Mutation.SignUp リゾルバーレスポンスハンドラー

API の他のリゾルバーで validateEmail 関数を再利用することが多いため、GraphQL フィールド間でアクセス先の ctx.args が変わるのを避けるとします。この場合、代わりに ctx.stash を使用して、signUp(input: Signup) 入力フィールド引数からの email 属性を保存できます。

リクエスト関数とレスポンス関数を置き換えてリゾルバーコードを更新してください。

```
export function request(ctx) {
    ctx.stash.email = ctx.args.input.email
    return {};
}

export function response(ctx) {
    return ctx.prev.result;
}
```

[作成] または [保存] を選択してリゾルバーを更新します。

ステップ 2: 関数を作成する

パイプラインリゾルバーページの [関数] セクションで、[関数の作成] をクリックして、[新しい関数の作成] をクリックします。リゾルバーページを経由せずに関数を作成することもできます。これを行うには、 AWS AppSync コンソールで関数ページに移動します。[関数の作成] ボタンを選択します。E メールが有効であり特定のドメインからのものかどうかをチェックする関数を作成しましょう。E メールが無効な場合、この関数はエラーを発生させます。それ以外の場合、渡された入力をすべて転送します。

NONE タイプのデータソースを作成したことを確認してください。[データソース名] リストでこのデータソースを選択します。関数名として、「validateEmail」と入力します。関数コード領域で、次のスニペットですべてを上書きします。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { email } = ctx.stash;
  const valid = util.matches(
    '^[a-zA-Z0-9_.+-]+@(?:(?:[a-zA-Z0-9-]+\.)?[a-zA-Z]+\.)?(myvaliddomain)\.com',
    email
  );
  if (!valid) {
    util.error(`"${email}" is not a valid email.`);
  }
```

```
  return { payload: { email } };
}

export function response(ctx) {
  return ctx.result;
}
```

選択内容を確認し、[作成] を選択します。これで、validateEmail 関数が作成されました。次のコードでこれらの手順を繰り返し、saveUser 関数を作成します (わかりやすくするため、NONE データソースを使用し、関数が実行された後にユーザーがシステムに保存されたものとしています。):

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  return ctx.prev.result;
}

export function response(ctx) {
  ctx.result.id = util.autoId();
  return ctx.result;
}
```

これで、saveUser 関数が作成されました。

ステップ 3: パイプラインリゾルバーに関数を追加する

先ほど作成したパイプラインリゾルバーには関数が自動的に追加されています。そうでない場合や、[関数] ページから関数を作成した場合は、signUp リゾルバーページの [関数を追加] をクリックして、関数をアタッチできます。validateEmail と saveUser の両方の関数をリゾルバーに追加します。validateEmail 関数は saveUser 関数の前に配置する必要があります。関数を追加するときには、上に移動および下に移動オプションを使用して関数の実行順序を並べ替えることができます。変更を確認し、[保存] を選択します。

ステップ 4: クエリを実行する

AWS AppSync コンソールで、クエリページに移動します。Explorer で、ミューテーションを使用していることを確認します。そうでない場合は、ドロップダウンリストから [Mutation] を選択し、[+] を選択します。以下のクエリを入力します。

```
mutation {
```

```
  signUp(input: {email: "nadia@myvaliddomain.com", username: "nadia"}) {
    id
    username
  }
}
```

以下のように返されます。

```
{
  "data": {
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "username": "nadia"
    }
  }
}
```

これで、パイプラインリゾルバーを使用して、ユーザーをサインアップし、入力 E メールを検証できました。

基本クエリの作成 (VTL)

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちらにある APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

GraphQL リゾルバーは、タイプのスキーマのフィールドをデータソースに接続します。リゾルバーは、リクエストが受理されるメカニズムです。 AWS AppSync は、コードを記述することなく、スキーマからリゾルバーを自動的に作成して接続したり、スキーマを作成して既存のテーブルからリゾルバーに接続したりできます。

AWS AppSync のリゾルバーは、JavaScript を使用して GraphQL 式をデータソースが使用できる形式に変換します。または、マッピングテンプレートを Apache Velocity Template Language (VTL) で記述すると、GraphQL 表現をデータソースで使用できる形式に変換できます。

このセクションでは、VTL を使用してリゾルバーを設定する方法について説明します。リゾルバーを記述するための入門チュートリアル形式のプログラミングガイドは、リゾルバーマッピングテンプレートプログラミングガイドにあり、プログラミング時に使用できるヘルパーユーティリティは、リ

ゾルバーマッピングテンプレートコンテキストリファレンスにあります。 AWS AppSync には、ゼロから編集またはオーサリングするときに使用できるテストおよびデバッグフローも組み込まれています。詳細については、「リゾルバーのテストとデバッグ」を参照してください。

前述のチュートリアルを使用する前に、このガイドに従うことをお勧めします。

このセクションでは、リゾルバーを作成する方法、ミューテーション用のリゾルバーを追加する方法、詳細設定を使用する方法について説明します。

最初のリゾルバーを作成する

前のセクションの例に従うと、最初のステップとして、ご使用の `Query` タイプに合ったリゾルバーを作成します。

Console

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   a. API ダッシュボードで、GraphQL API を選択します。
   b. サイドバーで [スキーマ] を選択します。
2. ページの右側には、[リゾルバー] というウィンドウがあります。このボックスには、ページの左側の [スキーマ] ウィンドウで定義されているタイプとフィールドのリストが含まれています。リゾルバーはフィールドにアタッチできます。たとえば、[クエリ] タイプで、`getTodos` フィールドの横にある [アタッチ] を選択します。
3. [リゾルバーの作成] ページで、「データソースを追加する」ガイドで作成したデータソースを選択します。[マッピングテンプレートの設定] ウィンドウでは、右側のドロップダウンリストを使用して汎用のリクエストおよびレスポンスのマッピングテンプレートを両方とも選択することも、独自のテンプレートを作成することもできます。

> **Note**
>
> リクエストマッピングテンプレートとレスポンスマッピングテンプレートの組み合わせをユニットリゾルバーと呼びます。ユニットリゾルバーは通常、機械的なオペレーションを実行するためのものであるため、データソース数が少ない単一のオペレーションのみに使用することをおすすめします。より複雑なオペレーションには、複数のデータソースで複数のオペレーションを連続して実行できるパイプラインリゾルバーの使用をお勧めします。
> リクエストマッピングテンプレートとレスポンスマッピングテンプレートの違いの詳細については、「ユニットリゾルバー」を参照してください。

パイプラインリゾルバーの使用に関する詳細については、「パイプラインリゾルバー」を参照してください。

4. 一般的なユースケースでは、 AWS AppSync コンソールには、データソースから項目を取得するために使用できる組み込みテンプレート (すべての項目クエリ、個々のルックアップなど) があります。たとえば、「スキーマの設計」で使用したスキーマのシンプルバージョンでは、getTodos にページ分割がなく、項目を一覧表示するためのリクエストマッピングテンプレートは次のようになっていました。

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
}
```

5. リクエストに関連付けるレスポンスマッピングテンプレートが常に必要になります。コンソールには、リストの値をパススルーする次のようなデフォルトのテンプレートがあります。

```
$util.toJson($ctx.result.items)
```

この例では、項目のリストに対する context オブジェクト ($ctx としてエイリアスが作成された) の形式は $context.result.items です。GraphQL オペレーションが 1 つの項目を返す場合、 になります$context.result。 AWS AppSync は、レスポンスを適切にフォーマットするために、前述の関数など、一般的なオペレーション用のヘルパー$util.toJson関数を提供します。関数の完全なリストについては、「リゾルバーのマッピングテンプレートのユーティリティリファレンス」を参照してください。

6. [リゾルバーを保存] を選択します。

API

1. CreateResolver API を呼び出して、リゾルバーオブジェクトを作成します。
2. UpdateResolver API を呼び出して、リゾルバーのフィールドを変更できます。

CLI

1. create-resolver コマンドを実行してリゾルバーを作成します。

このコマンドには次の 6 つのパラメータを入力する必要があります。

1. API の api-id。
2. スキーマ内で変更するタイプの type-name。このコンソールの例では、Query です。
3. タイプ内で変更するフィールドの field-name。このコンソールの例では、getTodos です。
4. 「データソースを追加する」ガイドで作成したデータソースの data-source-name。
5. request-mapping-template。リクエストの本文です。このコンソールの例では、次のようになります。

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
}
```

6. response-mapping-template。リクエストの本文です。このコンソールの例では、次のようになります。

```
$util.toJson($ctx.result.items)
```

コマンドの例は、次のようになります。

```
aws appsync create-resolver --api-id abcdefghijklmnopqrstuvwxyz --type-name
 Query --field-name getTodos --data-source-name TodoTable --request-mapping-
template "{ "version" : "2017-02-28", "operation" : "Scan", }" --response-
mapping-template ""$"util.toJson("$"ctx.result.items)"
```

出力は CLI に返されます。例を示します。

```
{
    "resolver": {
        "kind": "UNIT",
        "dataSourceName": "TodoTable",
        "requestMappingTemplate": "{ version : 2017-02-28, operation : Scan, }",
        "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Query/resolvers/getTodos",
        "typeName": "Query",
        "fieldName": "getTodos",
```

```
        "responseMappingTemplate": "$util.toJson($ctx.result.items)"
    }
}
```

2. リゾルバーのフィールドおよび/またはマッピングテンプレートを変更するには、update-resolver コマンドを実行します。

   api-id パラメータを除いて、create-resolver コマンドで使用されたパラメータは、update-resolver コマンドの新しい値で上書きされます。

ミューテーション用のリゾルバーの追加

次のステップは、ご使用の Mutation タイプに合ったリゾルバーを作成することです。

Console

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。

   a. API ダッシュボードで、GraphQL API を選択します。

   b. サイドバーで [スキーマ] を選択します。

2. [ミューテーション] タイプで、addTodo フィールドの横にある [アタッチ] を選択します。
3. [リゾルバーの作成] ページで、「データソースを追加する」ガイドで作成したデータソースを選択します。
4. このミューテーションでは DynamoDB に新しい項目を追加するため、[マッピングテンプレートの設定] ウィンドウでリクエストテンプレートを変更する必要があります。次のリクエストマッピングテンプレートを使用します。

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

5. AWS AppSync は、 addTodoフィールドで定義された引数を GraphQL スキーマから DynamoDB オペレーションに自動的に変換します。上記の例では、ミューテーションの引数で $ctx.args.id として渡された id のキーを使用して、レコードが DynamoDB に保存されます。渡した他のすべてのフィールド

は、`$util.dynamodb.toMapValuesJson($ctx.args)` を使用して自動的に DynamoDB 属性にマッピングされます。

このリゾルバーでは、次のレスポンスマッピングテンプレートを使用します。

```
$util.toJson($ctx.result)
```

AWS AppSync は、リゾルバーを編集するためのテストワークフローとデバッグワークフローもサポートしています。モック `context` オブジェクトを使用して、呼び出す前にテンプレートでの変換後の値を確認できます。また、クエリを実行する際にデータソースへのリクエストの実行全体をインタラクティブに表示することもできます。詳細については、「リゾルバーのテストとデバッグ」および「モニタリングとロギング」を参照してください。

6. [リゾルバーを保存] を選択します。

API

API では、[最初のリゾルバーを作成する] セクションのコマンドと、このセクションのパラメータの詳細を利用することで、これを行うこともできます。

CLI

CLI では、[最初のリゾルバーを作成する] セクションのコマンドと、このセクションのパラメータの詳細を利用することで、これを行うこともできます。

この時点で、高度なリゾルバーを使用していない場合は、「API の使用」で説明されているように GraphQL API の使用を開始できます。

高度なリゾルバー

「スキーマの設計」でのサンプルスキーマの構築の「高度な機能」セクションに従ってページ分割スキャンを行う場合は、代わりに `getTodos` フィールドに次のリクエストテンプレートを使用します。

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "limit": $util.defaultIfNull(${ctx.args.limit}, 20),
    "nextToken": $util.toJson($util.defaultIfNullOrBlank($ctx.args.nextToken, null))
}
```

このページ分割のユースケースでは、レスポンスマッピングに cursor (クライアントが次の開始ページを知るため) と結果セットの両方が含まれている必要があるため、レスポンスマッピングは単なるパススルーではありません。マッピングテンプレートは次のようになります。

```
{
    "todos": $util.toJson($context.result.items),
    "nextToken": $util.toJson($context.result.nextToken)
}
```

前述のレスポンスマッピングテンプレート内のフィールドは、`TodoConnection` 型で定義されているフィールドと一致している必要があります。

`Comments` テーブルがあり、`Todo` 型の comments フィールドを解決するリレーション (`[Comment]` の型を返す) の場合は、2 番目のテーブルに対してクエリを実行するマッピングテンプレートを使用できます。これを行うには、「データソースをアタッチする」で説明しているように `Comments` テーブルのデータソースを作成しておく必要があります。

> **Note**
>
> ここで 2 番目のテーブルに対するクエリオペレーションを使用しているのは、例を示すことのみを目的としています。代わりに、DynamoDB に対して別のオペレーションを使用できます。さらに、リレーションは GraphQL スキーマによって制御されるため、 AWS Lambda や Amazon OpenSearch Service などの別のデータソースからデータを取得できます。

Console

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   a. API ダッシュボードで、GraphQL API を選択します。
   b. サイドバーで [スキーマ] を選択します。
2. [Todo] タイプで、`comments` フィールドの横にある [アタッチ] を選択します。
3. [リゾルバーの作成] ページで、Comments テーブルのデータソースを選択します。クイックスタートガイドの Comments テーブルのデフォルト名は `AppSyncCommentTable` ですが、指定した名前によって異なる場合があります。
4. リクエストマッピングテンプレートに次のスニペットを追加します。

```
{
```

```
    "version": "2017-02-28",
    "operation": "Query",
    "index": "todoid-index",
    "query": {
        "expression": "todoid = :todoid",
        "expressionValues": {
            ":todoid": {
                "S": $util.toJson($context.source.id)
            }
        }
    }
}
```

5. context.source は、解決する現在のフィールドの親オブジェクトを参照します。この例では、source.id は個別の Todo オブジェクトを参照します。これはクエリ式に使用されます。

   次のようにパススルーのレスポンスマッピングテンプレートを使用できます。

```
$util.toJson($ctx.result.items)
```

6. [リゾルバーを保存] を選択します。
7. 最後に、コンソールの [スキーマ] ページで、リゾルバーを addComment フィールドにアタッチして Comments テーブルのデータソースを指定します。この例のリクエストマッピングテンプレートは、引数としてコメントされる特定の todoid を含む単純な PutItem ですが、次のように $utils.autoId() ユーティリティを使用してコメントに対して一意なソートキーを作成します。

```
{
    "version": "2017-02-28",
    "operation": "PutItem",
    "key": {
        "todoid": { "S": $util.toJson($context.arguments.todoid) },
        "commentid": { "S": "$util.autoId()" }
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

   次のようにパススルーレスポンステンプレートを使用します。

```
$util.toJson($ctx.result)
```

API

API では、[最初のリゾルバーを作成する] セクションのコマンドと、このセクションのパラメータの詳細を利用することで、これを行うこともできます。

CLI

CLI では、[最初のリゾルバーを作成する] セクションのコマンドと、このセクションのパラメータの詳細を利用することで、これを行うこともできます。

ダイレクト Lambda リゾルバー (VTL) を使用した VTL マッピングテンプレートの無効化

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

直接 Lambda リゾルバーを使用すると、 AWS Lambda データソースを使用するときに VTL マッピングテンプレートの使用を回避できます。 AWS AppSync は、Lambda 関数へのデフォルトのペイロードと、GraphQL タイプへの Lambda 関数のレスポンスからのデフォルトの変換を提供できます。リクエストテンプレート、レスポンステンプレート、またはどちらも指定できず、 AWS AppSync はそれに応じて処理します。

AWS AppSync が提供するデフォルトのリクエストペイロードとレスポンス変換の詳細については、「Direct Lambda リゾルバーリファレンス」を参照してください。 AWS Lambda データソースの設定と IAM 信頼ポリシーの設定の詳細については、「データソースのアタッチ」を参照してください。

ダイレクト Lambda リゾルバーを設定する

以下のセクションでは、Lambda データソースをアタッチし、Lambda リゾルバーをフィールドに追加する方法を示します。

Lambda データソースを追加する

ダイレクト Lambda リゾルバーをアクティブ化する前に、Lambda データソースを追加する必要があります。

Console

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   a. API ダッシュボードで、GraphQL API を選択します。
   b. サイドバーで [データソース] を選択します。
2. [データソースを作成] を選択します。
   a. データソース名を使用する場合、**myFunction** のようなデータソースの名前を入力します。
   b. [データソースタイプ] で [AWS Lambda 関数] を選択します。
   c. [リージョン] で、該当するリージョンを選択します。
   d. 関数 ARNを使用する場合は、ドロップダウンリストから Lambda 関数を選択します。関数名を検索するか、使用する関数の ARN を手動で入力します。
   e. 次に、新しい IAM ロールを作成するか (推奨)、`lambda:invokeFunction` への IAM アクセス許可を持つ既存のロールを選択します。データソースのアタッチセクションで説明しているように、既存のロールには信頼ポリシーが必要です。

      次に、リソースで操作を実行するために必要なアクセス許可を持つ IAM ポリシーの例を示します。

      JSON

```
{
     "Version": "2012-10-17",
     "Statement": [
         {
             "Effect": "Allow",
             "Action": [ "lambda:invokeFunction" ],
             "Resource": [
                 "arn:aws:lambda:us-
west-2:123456789012:function:myFunction",
                 "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:*"
```

```
            ]
        }
    ]
}
```

3. [サーバーの作成] ボタンを選択します。

CLI

1. create-data-source コマンドを実行してデータソースオブジェクトを作成します。

   このコマンドには次の 4 つのパラメータを入力する必要があります。

   1. API の api-id です。
   2. データソースの name。コンソールの例では、これはデータソース名です。
   3. データソースの type。コンソールの例では、これはAWS Lambda 関数です。
   4. コンソールの例では関数 ARN の lambda-config。

   > **Note**
   >
   > Region など、設定する必要のあるパラメータは他にもありますが、通常はデフォルトで CLI 設定値になります。

   コマンドの例は、次のようになります。

```
aws appsync create-data-source --api-id abcdefghijklmnopqrstuvwxyz
 --name myFunction --type AWS_LAMBDA --lambda-config
 lambdaFunctionArn=arn:aws:lambda:us-west-2:102847592837:function:appsync-
lambda-example
```

   出力は CLI に返されます。例を示します。

```
{
    "dataSource": {
        "dataSourceArn": "arn:aws:appsync:us-west-2:102847592837:apis/
abcdefghijklmnopqrstuvwxyz/datasources/myFunction",
        "type": "AWS_LAMBDA",
        "name": "myFunction",
```

```
        "lambdaConfig": {
            "lambdaFunctionArn": "arn:aws:lambda:us-
west-2:102847592837:function:appsync-lambda-example"
        }
    }
}
```

2. データソースの属性を変更するには、update-data-source コマンドを実行します。

   api-id パラメータを除いて、create-data-source コマンドで使用されているパラメータは、update-data-source コマンドの新しい値で上書きされます。

ダイレクト Lambda リゾルバをアクティブ化する

Lambda データソースを作成し、 AWS AppSync が関数を呼び出せるように適切な IAM ロールを設定したら、リゾルバーまたはパイプライン関数にリンクできます。

Console

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   a. API ダッシュボードで、GraphQL API を選択します。
   b. サイドバーで [スキーマ] を選択します。
2. [リゾルバー] セクションで、フィールドまたはオペレーションを選択し、[アタッチ] ボタンを選択します。
3. 新しいリゾルバーを作成するページで、ドロップダウンリストから Lambda 関数を選択します。
4. ダイレクト Lambda リゾルバーを利用するには、リクエストとレスポンスのマッピングテンプレートがマッピングテンプレートを設定するセクションで無効になっていることを確認します。
5. [リゾルバーの保存] ボタンを選択します。

CLI

- create-resolver コマンドを実行してリゾルバーを作成します。

  このコマンドには次の 6 つのパラメータを入力する必要があります。

  1. API の api-id。

2. スキーマでのタイプの `type-name`。
3. スキーマでのフィールドの `field-name`。
4. `data-source-name`、または Lambda 関数の名前。
5. リクエストの本文である `request-mapping-template`。コンソールの例では、これは無効になっています。

```
" "
```

6. レスポンスの本文である `response-mapping-template`。コンソールの例では、これも無効になっています。

```
" "
```

コマンドの例は、次のようになります。

```
aws appsync create-resolver --api-id abcdefghijklmnopqrstuvwxyz --type-name
 Subscription --field-name onCreateTodo --data-source-name LambdaTest --request-
mapping-template " " --response-mapping-template " "
```

出力は CLI に返されます。例を示します。

```
{
    "resolver": {
        "resolverArn": "arn:aws:appsync:us-west-2:102847592837:apis/
abcdefghijklmnopqrstuvwxyz/types/Subscription/resolvers/onCreateTodo",
        "typeName": "Subscription",
        "kind": "UNIT",
        "fieldName": "onCreateTodo",
        "dataSourceName": "LambdaTest"
    }
}
```

マッピングテンプレートを無効にすると、 AWS AppSyncで発生する追加の動作がいくつかあります。

- マッピングテンプレートを無効にすると、Direct Lambda リゾルバーリファレンスで指定されたデフォルトのデータ変換を受け入れる AWS AppSync ことを に知らせます。

- リクエストマッピングテンプレートを無効にすると、Lambda データソースは、Contextオブジェクト全体で構成されるペイロードを受け取ります。
- レスポンスマッピングテンプレートを無効にすると、リクエストマッピングテンプレートのバージョン、またはリクエストマッピングテンプレートも無効になっているかどうかに応じて、Lambda 呼び出しの結果が変換されます。

AWS AppSync (VTL) でのリゾルバーのテストとデバッグ

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync は、データソースに対して GraphQL フィールドでリゾルバーを実行します。「リゾルバーのマッピングテンプレートの概要」で説明しているように、リゾルバーはテンプレート作成言語を使用してデータソースと通信します。これにより、ユーザーは動作をカスタマイズでき、データソースと通信する前後にロジックおよび条件を適用できます。リゾルバーを記述するための入門者向けのチュートリアル形式プログラミングガイドについては、「リゾルバーのマッピングテンプレートのプログラミングガイド」を参照してください。

デベロッパーがこれらのリゾルバーを記述、テスト、デバッグできるように、 AWS AppSync コンソールには、モックデータを含む GraphQL リクエストとレスポンスを個々のフィールドリゾルバーに作成するためのツールも用意されています。さらに、 AWS AppSync コンソールでクエリ、ミューテーション、サブスクリプションを実行し、リクエスト全体の Amazon CloudWatch からの詳細なログストリームを表示できます。ログストリームにはデータソースからの結果も含まれています。

モックデータを使用したテスト

GraphQL リゾルバーが呼び出されるときに、そのリゾルバーには、リクエストに関する情報を含む `context` オブジェクトが含まれています。このオブジェクトには、クライアントからの引数、ID 情報、および親 GraphQL フィールドからのデータが含まれています。また、データソースからの結果も含まれていて、それをレスポンステンプレートで使用できます。この構造体およびプログラミング時に使用可能なヘルパーユーティリティの詳細については、「リゾルバーのマッピングテンプレートのコンテキストリファレンス」を参照してください。

リゾルバーを記述または編集する場合に、モックまたはテストコンテキストオブジェクトをコンソールエディタに渡すことができます。これにより、実際にデータソースに対して実行することなく、リクエストとレスポンスの両方のテンプレートでどのように評価されるかを確認できます。例えば、テストの `firstname: Shaggy` 引数を渡して、テンプレートのコードで `$ctx.args.firstname` を使用している場合にその引数がどのように評価されるかを確認できます。任意のユーティリティヘルパー (`$util.autoId()`、`util.time.nowISO8601()` など) での評価をテストすることもできます。

リゾルバーのテスト

この例では、 AWS AppSync コンソールを使用してリゾルバーをテストします。

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   a. API ダッシュボードで、GraphQL API を選択します。
   b. サイドバーで [スキーマ] を選択します。
2. まだ行っていない場合は、タイプの下のフィールドの横にある [アタッチ] を選択してリゾルバーを追加します。

   完全なリゾルバーをビルドする方法の詳細については、「リゾルバーを設定する」を参照してください。

   それ以外の場合は、すでにフィールドにあるリゾルバーを選択してください。
3. [リゾルバーを編集] ページの上部で、[テストコンテキストを選択] を選択し、[新しいコンテキストを作成] を選択します。
4. サンプルコンテキストオブジェクトを選択するか、下の実行コンテキストウィンドウに JSON を手動で入力します。
5. テキストコンテキスト名を入力します。
6. [保存] ボタンを選択します。
7. [リゾルバーを編集] ページの上部にある [テストを実行] を選択します。

より現実的な例として、オブジェクトに対して自動 ID 生成を使用して Amazon DynamoDB に保存する Dog の GraphQL タイプを保存するアプリケーションがあるとします。また、一部の値を GraphQL ミューテーションの引数から書き込み、レスポンスが特定の 1 人のユーザーにのみ表示されるようにします。スキーマは次のようになります。

```
type Dog {
```

```
  breed: String
  color: String
}

type Mutation {
  addDog(firstname: String, age: Int): Dog
}
```

addDog ミューテーションに対してリゾルバーを追加するときに、次のようなコンテキストオブジェクトを入力できます。次の例には、クライアントから引数として name と age があり、identity オブジェクトに入力されている username があります。

```
{
    "arguments" : {
        "firstname": "Shaggy",
        "age": 4
    },
    "source" : {},
    "result" : {
        "breed" : "Miniature Schnauzer",
        "color" : "black_grey"
    },
    "identity": {
        "sub" : "uuid",
        "issuer" : " https://cognito-idp.{region}.amazonaws.com/{userPoolId}",
        "username" : "Nadia",
        "claims" : { },
        "sourceIp" :[  "x.x.x.x" ],
        "defaultAuthStrategy" : "ALLOW"
    }
}
```

これを、以下のリクエストとレスポンスのマッピングテンプレートを使用してテストします。

リクエストテンプレート

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : { "S" : "$util.autoId()" }
    },
```

```
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

レスポンステンプレート

```
#if ($context.identity.username == "Nadia")
  $util.toJson($ctx.result)
#else
  $util.unauthorized()
#end
```

評価後のテンプレートには、テストコンテキストオブジェクトからのデータと、`$util.autoId()` から生成された値があります。また、`username` を `Nadia` 以外の値に変更した場合は、認可チェックが失敗するため、結果は返されません。きめ細かなアクセス制御の詳細については、「認可のユースケース」を参照してください。

AWS AppSync の APIs を使用したマッピングテンプレートのテスト

`EvaluateMappingTemplate` API コマンドを使用して、モックデータを使用してマッピングテンプレートをリモートでテストできます。コマンドを使い始めるには、ポリシーに `appsync:evaluateMappingTemplate` アクセス許可を追加していることを確認してください。例:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateMappingTemplate",
            "Resource": "arn:aws:appsync:<region>:<account>:*"
        }
    ]
}
```

AWS CLI または AWS SDK を使用してコマンドを活用できます。例えば、前のセクションから、Dog スキーマとそのリクエスト/レスポンスマッピングテンプレートを見てみましょう。ローカルステーションの CLI を使用して、リクエストテンプレートを `request.vtl` という名前のファイ

ルに保存し、context オブジェクトを context.json という名前のファイルに保存します。シェルから次のコマンドを実行します。

```
aws appsync evaluate-mapping-template --template file://request.vtl --context file://
context.json
```

このコマンドは、次のレスポンスを返します。

```
{
  "evaluationResult": "{\n    \"version\" : \"2017-02-28\",\n
 \"operation\" : \"PutItem\",\n    \"key\" : {\n        \"id\" : { \"S\" :
 \"afcb4c85-49f8-40de-8f2b-248949176456\" }\n    },\n    \"attributeValues\" :
 {\"firstname\":{\"S\":\"Shaggy\"},\"age\":{\"N\":4}}\n}\n"
}
```

evaluationResult には、提供されたテンプレートを提供された context とともにテストした結果が含まれています。 AWS SDKs を使用してテンプレートをテストすることもできます。 AWS SDK for JavaScript V2 を使用した例を次に示します。

```
const AWS = require('aws-sdk')
const client = new AWS.AppSync({ region: 'us-east-2' })

const template = fs.readFileSync('./request.vtl', 'utf8')
const context = fs.readFileSync('./context.json', 'utf8')

client
  .evaluateMappingTemplate({ template, context })
  .promise()
  .then((data) => console.log(data))
```

SDK を使用すると、お気に入りのテストスイートのテストを簡単に組み込んで、テンプレートの動作を検証できます。Jest テストフレームワークを使用してテストを作成することをお勧めしますが、どのテストスイートでも問題ありません。次のスニペットは、仮説検証の実行を示しています。評価レスポンスは有効な JSON であることを想定しているので、JSON.parseを使用して文字列レスポンスから JSON を取得することに注意してください。

```
const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })
```

```
test('request correctly calls DynamoDB', async () => {
  const template = fs.readFileSync('./request.vtl', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateMappingTemplate({ template,
 context }).promise()
  const result = JSON.parse(response.evaluationResult)

  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})
```

これにより、次のような結果が得られます。

```
Ran all test suites.
> jest

PASS ./index.test.js
# request correctly calls DynamoDB (543 ms)

Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 1.511 s, estimated 2 s
```

ライブクエリのデバッグ

本番アプリケーションをデバッグするためのend-to-endのテストとログ記録に代わるものはありません。 AWS AppSync を使用すると、Amazon CloudWatch を使用してエラーとリクエストの詳細全体をログに記録することができます。さらに、 AWS AppSync コンソールを使用して、GraphQL クエリ、ミューテーション、サブスクリプションをテストし、各リクエストのライブストリームログデータをクエリエディタに戻し、リアルタイムでデバッグできます。サブスクリプションに関して表示されるログは接続時の情報です。

これを実行するには、「モニタリングとロギング」で説明しているように、Amazon CloudWatch Logs を事前に有効にしておく必要があります。次に、 AWS AppSync コンソールでクエリタブを選択し、有効な GraphQL クエリを入力します。右下のセクションで、[ログ] ウィンドウをクリックしてドラッグし、ログビューを開きます。ページの上部にある再生矢印アイコンを選択して GraphQL クエリを実行します。しばらくすると、そのオペレーションのリクエストとレスポンスの完全なログが、このセクションにストリーミングされ、コンソールで表示できます。

AWS AppSync (VTL) でのパイプラインリゾルバーの設定と使用

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync は GraphQL フィールドでリゾルバーを実行します。場合によっては、アプリケーションは 1 つの GraphQL フィールドを解決するために複数のオペレーションを実行する必要があります。パイプラインリゾルバーを使用すると、開発者は関数と呼ばれるオペレーションを構成して順番に実行できます。パイプラインリゾルバーは、たとえば、フィールドのデータを取得する前に承認チェックを実行する必要があるアプリケーションに便利です。

パイプラインリゾルバーは、Before マッピングテンプレート、After マッピングテンプレート、関数のリストで構成されています。各関数には、データソースに対して実行されるリクエストおよびレスポンスマッピングテンプレートがあります。パイプラインリゾルバーは実行をリスト内の関数に委任するため、いずれのデータソースにもリンクされていません。ユニットリゾルバーと関数は、データソースに対してオペレーションを実行するプリミティブです。詳細については、「リゾルバーのマッピングテンプレートの概要」を参照してください。

ステップ 1: パイプラインリゾルバーを作成する

AWS AppSync コンソールで、スキーマページに移動します。

以下のスキーマを保存します。

```
schema {
    query: Query
    mutation: Mutation
}

type Mutation {
    signUp(input: Signup): User
}

type Query {
    getUser(id: ID!): User
}

input Signup {
    username: String!
```

```
    email: String!
}

type User {
    id: ID!
    username: String
    email: AWSEmail
}
```

パイプラインリゾルバーをミューテーションタイプの SignUp フィールドに書き込みます。右側の [ミューテーション] タイプに入力し、[signUp ミューテーション] フィールドの横の [アタッチ] を選択します。[リゾルバーの作成] ページで、[アクション]、[ランタイム更新] の順にクリックします。[Pipeline Resolver]、[VTL] の順に選択してから、[更新] を選択します。このページには、Before マッピングテンプレートテキスト領域、関数セクション、After マッピングテンプレートテキスト領域の 3 つのセクションが表示されています。

パイプラインリゾルバーは、最初に E メールアドレスの入力を検証してからシステムにユーザーを保存することで、ユーザーをサインアップします。E メールの検証を validateEmail 関数内にカプセル化し、ユーザーの保存を saveUser 関数内にカプセル化します。validateEmail 関数が最初に実行され、E メールが有効であれば、saveUser 関数が実行されます。

実行フローは以下のようになります。

1. Mutation.signUp リゾルバーのリクエストマッピングテンプレート
2. validateEmail 関数
3. saveUser 関数
4. Mutation.signUp リゾルバーのレスポンスマッピングテンプレート

API の他のリゾルバーで validateEmail 関数を再利用することが多いため、GraphQL フィールド間でアクセス先の `$ctx.args` が変わるのを避けるとします。この場合、代わりに `$ctx.stash` を使用して、`signUp(input: Signup)` 入力フィールド引数からの email 属性を保存できます。

BEFORE マッピングテンプレート:

```
## store email input field into a generic email key
$util.qr($ctx.stash.put("email", $ctx.args.input.email))
{}
```

コンソールから、デフォルトのパススルー AFTER マッピングテンプレートを使用できます。

```
$util.toJson($ctx.result)
```

[作成] または [保存] を選択してリゾルバーを更新します。

ステップ 2: 関数を作成する

パイプラインリゾルバーページの [関数] セクションで、[関数の作成] をクリックして、[新しい関数の作成] をクリックします。リゾルバーページを経由せずに関数を作成することもできます。これを行うには、 AWS AppSync コンソールで関数ページに移動します。[関数の作成] ボタンを選択します。E メールが有効であり特定のドメインからのものかどうかをチェックする関数を作成しましょう。E メールが無効な場合、この関数はエラーを発生させます。それ以外の場合、渡された入力をすべて転送します。

新しい関数ページで、[アクション] を選択して、[ランタイムを更新] を選択します。[VTL] を選択してから、[更新] を選択します。NONE 型のデータソースが作成されたことを確認してください。[データソース名] リストでこのデータソースを選択します。[関数名] に `validateEmail` と入力します。関数コード 領域で、次のスニペットですべてを上書きします。

```
#set($valid = $util.matches("^[a-zA-Z0-9_.+-]+@(?:(?:[a-zA-Z0-9-]+\.)?[a-zA-Z]+\.)?
(myvaliddomain)\.com", $ctx.stash.email))
#if (!$valid)
    $util.error("$ctx.stash.email is not a valid email.")
#end
{
    "payload": { "email": $util.toJson(${ctx.stash.email}) }
}
```

レスポンスマッピングテンプレートにこれを貼り付けます。

```
$util.toJson($ctx.result)
```

変更内容を確認し、[作成] を選択します。これで、validateEmail 関数が作成されました。次のリクエストマッピングテンプレートとレスポンスマッピングテンプレートでこれらの手順を繰り返し、saveUser 関数を作成します (わかりやすくするため、NONE データソースを使用し、関数が実行された後にユーザーがシステムに保存されたものとしています。):

リクエストマッピングテンプレート:

```
## $ctx.prev.result contains the signup input values. We could have also
## used $ctx.args.input.
```

```
{
    "payload": $util.toJson($ctx.prev.result)
}
```

レスポンスマッピングテンプレート

```
## an id is required so let's add a unique random identifier to the output
$util.qr($ctx.result.put("id", $util.autoId()))
$util.toJson($ctx.result)
```

これで、saveUser 関数が作成されました。

ステップ 3: パイプラインリゾルバーに関数を追加する

先ほど作成したパイプラインリゾルバーには関数が自動的に追加されています。そうでない場合、または [関数] ページで関数を作成した場合は、リゾルバーページの [関数を追加] をクリックして、それらの関数をアタッチできます。validateEmail と saveUser の両方の関数をリゾルバーに追加します。validateEmail 関数は saveUser 関数の前に配置する必要があります。関数を追加するときには、上に移動および下に移動オプションを使用して関数の実行順序を並べ替えることができます。変更を確認し、[保存] を選択します。

ステップ 4: クエリを実行する

AWS AppSync コンソールで、クエリページに移動します。Explorer で、ミューテーションを使用していることを確認します。そうでない場合は、ドロップダウンリストから [Mutation] を選択し、[+] を選択します。以下のクエリを入力します。

```
mutation {
  signUp(input: {
    email: "nadia@myvaliddomain.com"
    username: "nadia"
  }) {
    id
    email
  }
}
```

以下のように返されます。

```
{
  "data": {
```

```
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "email": "nadia@myvaliddomain.com"
    }
  }
}
```

これで、パイプラインリゾルバーを使用して、ユーザーをサインアップし、入力 E メールを検証できました。パイプラインリゾルバーに焦点を当てたより完全なチュートリアルについては、「チュートリアル: パイプラインリゾルバー」を参照してください。

AWS CDK での AWS AppSync API の使用

> **Tip**
>
> CDK を使用する前に、CDK の公式ドキュメントと CDK AWS AppSyncリファレンスを確認することをお勧めします。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html
> また、AWS CLI と NPM のインストールがシステム上で動作していることを確認することをお勧めします。

このセクションでは、DynamoDB テーブルに項目を追加したり、テーブルから項目を取得したりできる簡単な CDK アプリケーションを作成します。これは、[スキーマの設計]、[データソースのアタッチ]、[リゾルバーの設定 (JavaScript)] セクションのコードの一部を使用したクイックスタートの例です。

CDK プロジェクトのセットアップ

> **Warning**
>
> 環境によっては、これらの手順が完全に正確ではない場合があります。システムに必要なユーティリティ、 AWS サービスとのインターフェイス方法、適切な設定がインストールされていることを前提としています。

最初のステップは AWS CDK のインストールです。CLI で、次のコマンドを入力できます。

```
npm install -g aws-cdk
```

次に、プロジェクトディレクトリを作成して、そこに移動する必要があります。ディレクトリを作成して、そこに移動するコマンドのセット例は次のとおりです。

```
mkdir example-cdk-app
cd example-cdk-app
```

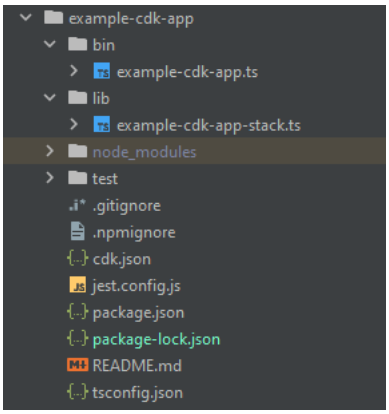
次に、アプリを作成する必要があります。私たちのサービスは主に TypeScript を使用しています。プロジェクトディレクトリで次のコマンドを入力します。

```
cdk init app --language typescript
```

これを行うと、CDK アプリとその初期化ファイルがインストールされます。

```
Initializing a new git repository...
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Executing npm install...
✅All done!
```

プロジェクト構造は次のようになります。



いくつかの重要なディレクトリがあるのがわかります。

- bin: 最初の bin ファイルによってアプリが作成されます。このガイドではこれについては触れません。

- `lib`: lib ディレクトリにはスタックファイルが含まれます。スタックファイルは個々の実行単位と考えることができます。コンストラクトはスタックファイル内にあります。基本的に、これらはアプリケーションのデプロイ AWS CloudFormation 時に でスピンアップされるサービスのリソースです。ほとんどのコーディングはここで行われます。
- `node_modules`: このディレクトリは NPM によって作成され、`npm` コマンドを使用してインストールしたすべてのパッケージ依存関係が含まれます。

最初のスタックファイルには次のような内容が含まれる場合があります。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
// import * as sqs from 'aws-cdk-lib/aws-sqs';

export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // The code that defines your stack goes here

    // example resource
    // const queue = new sqs.Queue(this, 'ExampleCdkAppQueue', {
    //   visibilityTimeout: cdk.Duration.seconds(300)
    // });
  }
}
```

これはアプリでスタックを作成するためのボイラープレートコードです。この例では、コードのほとんどはこのクラスのスコープ内にあります。

スタックファイルがアプリ内にあることを確認し、アプリのディレクトリからターミナルで以下のコマンドを実行します。

```
cdk ls
```

スタックが一覧表示されます。表示されない場合は、手順をもう一度実行するか、公式ドキュメントでヘルプを確認する必要があります。

デプロイする前にコードの変更をビルドする場合は、ターミナルでいつでも以下のコマンドを実行できます。

```
npm run build
```

また、デプロイ前に変更を確認します。

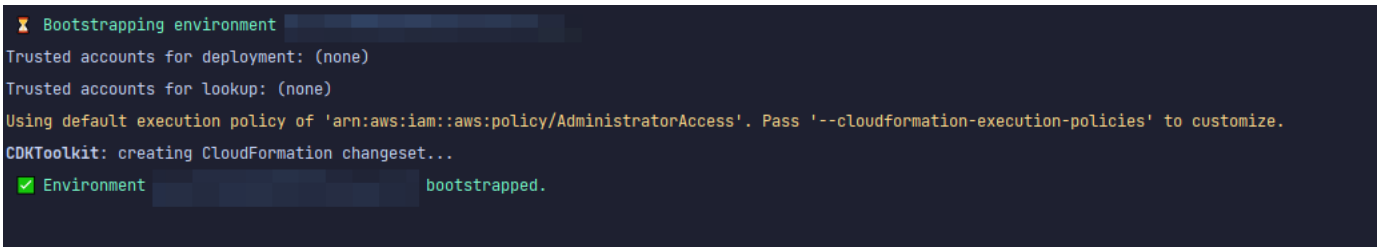
```
cdk diff
```

スタックファイルにコードを追加する前に、ブートストラップを実行します。ブートストラップにより、アプリがデプロイされる前に CDK にリソースをプロビジョニングできます。このプロセスの詳細については、こちらを参照してください。ブートストラップを作成するには、以下のコマンドを実行します。

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

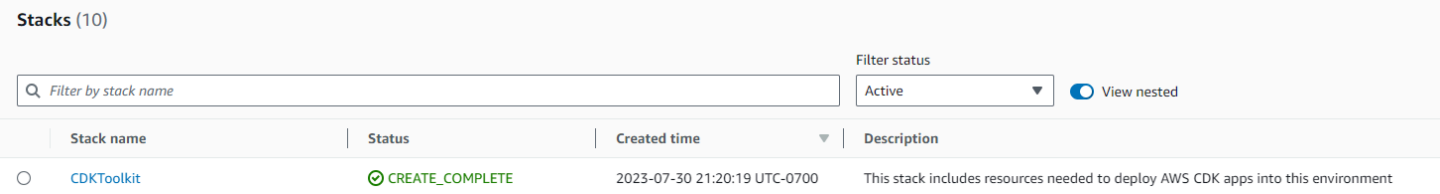
> **Tip**
>
> このステップでは、アカウントに複数の IAM アクセス許可が必要です。これらのアクセス許可がないと、ブートストラップは拒否されます。拒否された場合、ブートストラップが生成する S3 バケットなど、ブートストラップが原因で発生した不完全なリソースを削除しなければならない場合があります。

ブートストラップは複数のリソースを起動します。最終メッセージは次のようになります。

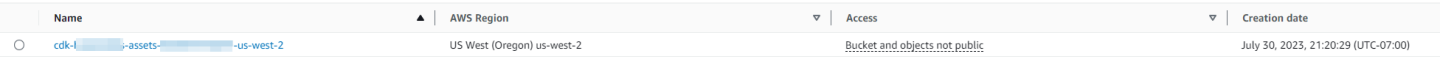


これは各リージョンのアカウントごとに 1 回行われるため、頻繁に行う必要はありません。ブートストラップの主なリソースは、 AWS CloudFormation スタックと Amazon S3 バケットです。

Amazon S3 バケットは、ファイルとデプロイの実行に必要なアクセス許可を付与する IAM ロールを保存するために使用されます。必要なリソースは、ブートストラップ AWS CloudFormation スタックと呼ばれる スタックで定義され、通常は と呼ばれますCDKToolkit。他の AWS CloudFormation スタックと同様に、デプロイされると AWS CloudFormation コンソールに表示されます。



バケットについても同じことが言えます。



スタックファイルで必要なサービスをインポートするには、以下のコマンドを使用できます。

```
npm install aws-cdk-lib # V2 command
```

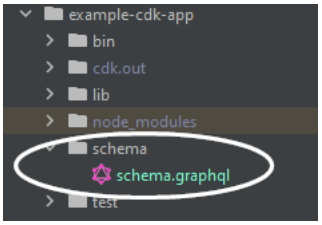
> **Tip**
>
> V2 に問題がある場合は、V1 コマンドを使用して個々のライブラリをインストールできます。
>
> ```
> npm install @aws-cdk/aws-appsync @aws-cdk/aws-dynamodb
> ```
>
> V1 は廃止されたため、これはお勧めしません。

CDK プロジェクトを作成する - スキーマ

これで、コードの実装を開始できます。まず、スキーマを作成する必要があります。アプリ内で `.graphql` ファイルを作成するだけです。

```
mkdir schema
touch schema.graphql
```

この例では、`schema.graphql` を含む `schema` という最上位ディレクトリを含めました。



スキーマの中に、簡単な例を含めてみましょう。

```
input CreatePostInput {
    title: String
    content: String
}

type Post {
    id: ID!
    title: String
    content: String
}

type Mutation {
    createPost(input: CreatePostInput!): Post
}

type Query {
    getPost: [Post]
}
```

スタックファイルに戻って、次の import ディレクティブが定義されていることを確認する必要があります。

```
import * as cdk from 'aws-cdk-lib';
import * as appsync from 'aws-cdk-lib/aws-appsync';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
import { Construct } from 'constructs';
```

クラス内に、GraphQL API を作成して `schema.graphql` ファイルに接続するコードを追加します。

```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // makes a GraphQL API
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });
  }
}
```

また、GraphQL URL、API キー、リージョンを出力するコードもいくつか追加します。

```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Makes a GraphQL API construct
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });

    // Prints out URL
    new cdk.CfnOutput(this, "GraphQLAPIURL", {
      value: api.graphqlUrl
    });

    // Prints out the AppSync GraphQL API key to the terminal
    new cdk.CfnOutput(this, "GraphQLAPIKey", {
      value: api.apiKey || ''
    });

    // Prints out the stack region to the terminal
    new cdk.CfnOutput(this, "Stack Region", {
      value: this.region
    });
  }
}
```

この時点で、アプリを再度デプロイします。

```
cdk deploy
```

こちらがその結果です。

```
ExampleCdkAppStack: deploying... [1/1]
ExampleCdkAppStack: creating CloudFormation changeset...

 ✅ ExampleCdkAppStack

✨ Deployment time: 16.13s

Outputs:
ExampleCdkAppStack.GraphQLAPIKey = 
ExampleCdkAppStack.GraphQLAPIURL = https://                                    /graphql
ExampleCdkAppStack.StackRegion = us-west-2
Stack ARN:
arn:aws:cloudformation:

✨ Total time: 22s
```

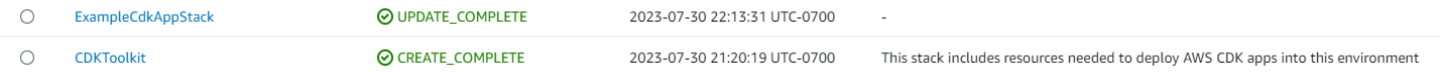
この例は成功したようですが、 AWS AppSync コンソールを確認して確認しましょう。

API が作成されたようです。次に、API に添付されているスキーマを確認します。

Schema

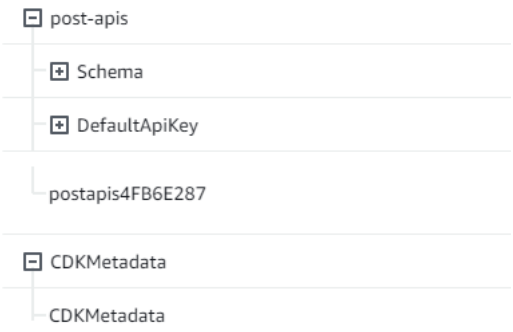
```
1  input CreatePostInput {
2    title: String
3    date: AWSDateTime
4  }
5
6  type Post {
7    id: ID!
8    title: String
9    date: AWSDateTime
10 }
11
12 type Mutation {
13   createPost(input: CreatePostInput!): Post
14 }
15
16 type Query {
17   getPost: [Post]
18 }
```

これはスキーマコードと一致しているようなので、成功です。メタデータの観点からこれを確認するもう 1 つの方法は、 AWS CloudFormation スタックを確認することです。



CDK アプリをデプロイすると、ブートストラップなどのリソース AWS CloudFormation がスピンアップされます。アプリ内の各スタックは、 AWS CloudFormation スタックと 1:1 でマッピングさ

れます。スタックコードに戻ると、スタック名はクラス名 ExampleCdkAppStack から取得されています。作成したリソースを確認でき、GraphQL API コンストラクトの命名規則とも一致しています。



CDK プロジェクトの実装 - データソース

次に、データソースを追加する必要があります。この例では DynamoDB テーブルを使用します。stack クラス内に、新しいテーブルを作成するコードをいくつか追加します。

```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Makes a GraphQL API construct
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });

    //creates a DDB table
    const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
      partitionKey: {
        name: 'id',
        type: dynamodb.AttributeType.STRING,
      },
    });

    // Prints out URL
    new cdk.CfnOutput(this, "GraphQLAPIURL", {
      value: api.graphqlUrl
    });

    // Prints out the AppSync GraphQL API key to the terminal
    new cdk.CfnOutput(this, "GraphQLAPIKey", {
```

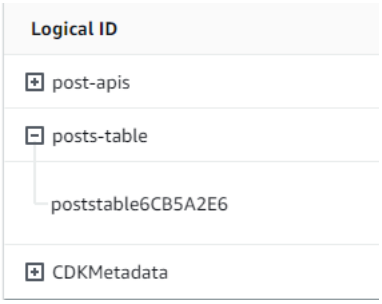
```
      value: api.apiKey || ''
    });

    // Prints out the stack region to the terminal
    new cdk.CfnOutput(this, "Stack Region", {
      value: this.region
    });
  }
}
```

この時点で、もう一度デプロイしてみましょう。

```
cdk deploy
```

DynamoDB コンソールで新しいテーブルを確認する必要があります。

スタック名は正しく、テーブル名はコードと一致しています。 AWS CloudFormation スタックを再度チェックすると、新しいテーブルが表示されます。



CDK プロジェクトを作成する - リゾルバー

この例では、2 つのリゾルバーを使用します。1 つはテーブルのクエリ用、もう 1 つはテーブルへの追加用です。パイプラインリゾルバーを使用しているため、それぞれ 1 つの関数を持つ 2 つのパイプラインリゾルバーを宣言する必要があります。このクエリでは、次のコードを追加します。

```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Makes a GraphQL API construct
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
```

```
    });

    //creates a DDB table
    const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
      partitionKey: {
        name: 'id',
        type: dynamodb.AttributeType.STRING,
      },
    });

    // Creates a function for query
    const add_func = new appsync.AppsyncFunction(this, 'func-get-post', {
      name: 'get_posts_func_1',
      api,
      dataSource: api.addDynamoDbDataSource('table-for-posts', add_ddb_table),
      code: appsync.Code.fromInline(`
          export function request(ctx) {
          return { operation: 'Scan' };
          }

          export function response(ctx) {
          return ctx.result.items;
          }
  `),
      runtime: appsync.FunctionRuntime.JS_1_0_0,
    });

    // Creates a function for mutation
    const add_func_2 = new appsync.AppsyncFunction(this, 'func-add-post', {
      name: 'add_posts_func_1',
      api,
      dataSource: api.addDynamoDbDataSource('table-for-posts-2', add_ddb_table),
      code: appsync.Code.fromInline(`
          export function request(ctx) {
            return {
            operation: 'PutItem',
            key: util.dynamodb.toMapValues({id: util.autoId()}),
            attributeValues: util.dynamodb.toMapValues(ctx.args.input),
            };
          }

          export function response(ctx) {
            return ctx.result;
          }
```

```
      `),
      runtime: appsync.FunctionRuntime.JS_1_0_0,
    });

    // Adds a pipeline resolver with the get function
    new appsync.Resolver(this, 'pipeline-resolver-get-posts', {
      api,
      typeName: 'Query',
      fieldName: 'getPost',
      code: appsync.Code.fromInline(`
          export function request(ctx) {
          return {};
          }

          export function response(ctx) {
          return ctx.prev.result;
          }
  `),
      runtime: appsync.FunctionRuntime.JS_1_0_0,
      pipelineConfig: [add_func],
    });

    // Adds a pipeline resolver with the create function
    new appsync.Resolver(this, 'pipeline-resolver-create-posts', {
      api,
      typeName: 'Mutation',
      fieldName: 'createPost',
      code: appsync.Code.fromInline(`
          export function request(ctx) {
          return {};
          }

          export function response(ctx) {
          return ctx.prev.result;
          }
  `),
      runtime: appsync.FunctionRuntime.JS_1_0_0,
      pipelineConfig: [add_func_2],
    });

    // Prints out URL
    new cdk.CfnOutput(this, "GraphQLAPIURL", {
      value: api.graphqlUrl
    });
```

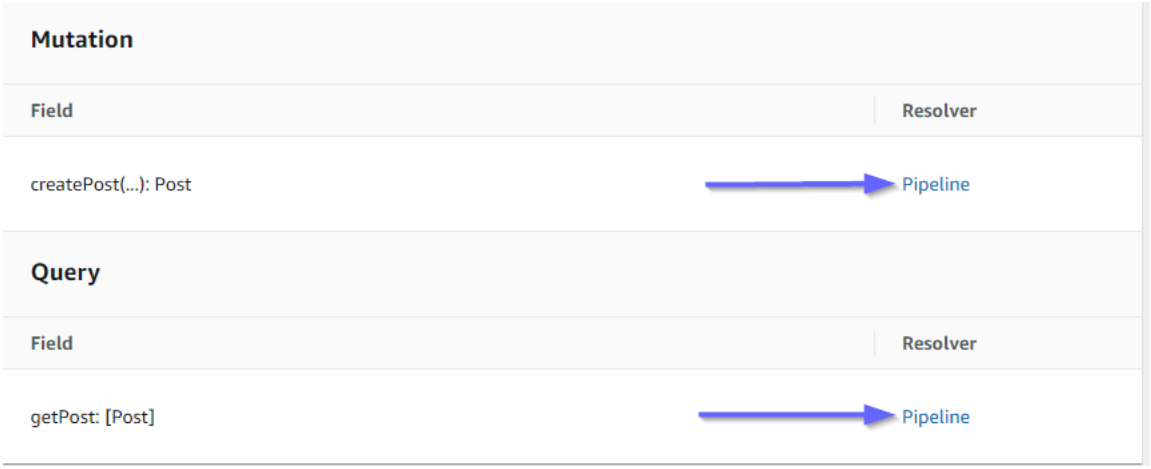
```
    // Prints out the AppSync GraphQL API key to the terminal
    new cdk.CfnOutput(this, "GraphQLAPIKey", {
      value: api.apiKey || ''
    });

    // Prints out the stack region to the terminal
    new cdk.CfnOutput(this, "Stack Region", {
      value: this.region
    });
  }
}
```

このスニペットでは、func-add-post という関数がアタッチされた pipeline-resolver-create-posts というパイプラインリゾルバーを追加しました。これがテーブルに Posts を追加するコードです。もう一方のパイプラインリゾルバーは pipeline-resolver-get-posts と呼ばれ、テーブルに追加された Posts を取得する func-get-post と呼ばれる関数を持ちます。

これをデプロイして AWS AppSync サービスに追加します。

```
cdk deploy
```

AWS AppSync コンソールで、GraphQL API にアタッチされているかどうかを確認します。



正しく接続されているようです。コードでは、これらのリゾルバーは両方とも、作成した GraphQL API にアタッチされていました (リゾルバーと関数の両方に存在する api props 値で示されます)。GraphQL APIでは、リゾルバーをアタッチしたフィールドも props で指定されていました (各リゾルバーで typename および fieldname props によって定義されます)。

リゾルバーのコンテンツが正しいかどうか、pipeline-resolver-get-posts を手始めに見てみましょう。

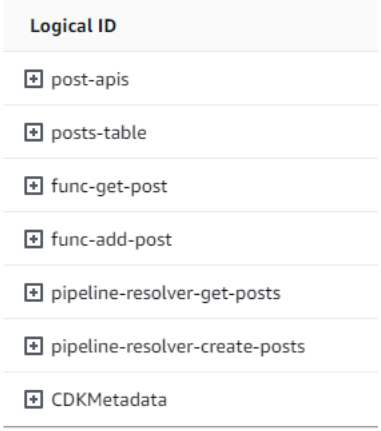
before ハンドラーと after ハンドラーは code props の値と一致しています。また、add_posts_func_1 という関数がリゾルバーにアタッチした関数の名前と一致していることもわかります。

この関数のコードコンテンツを見てみましょう。

これは add_posts_func_1 関数の code props と一致します。クエリが正常にアップロードされたので、クエリを確認してみましょう。

これらもコードと一致しています。get_posts_func_1 を見てみましょう。

すべてが正常に見えます。メタデータの観点からこれを確認するには、スタックをもう一度 AWS CloudFormation にチェックインします。



次に、いくつかのリクエストを実行して、このコードをテストする必要があります。

CDK プロジェクトの実装 - リクエスト

AWS AppSync コンソールでアプリケーションをテストするために、1 つのクエリと 1 つのミューテーションを作成しました。

```
query MyQuery {
  getPost {
    id
    date
    title
  }
}

mutation MyMutation {
  createPost(input: {date: "1970-01-01T12:30:00.000Z", title: "first post"}) {
    date
    id
    title
  }
}

```

MyMutation には、引数 1970-01-01T12:30:00.000Z と first post を含む createPost オペレーションが含まれています。渡した date および title とともに、自動生成された id 値が返されます。ミューテーションを実行すると、以下の結果が得られます。

```
{
  "data": {
    "createPost": {
      "date": "1970-01-01T12:30:00.000Z",
      "id": "4dc1c2dd-0aa3-4055-9eca-7c140062ada2",
      "title": "first post"
    }
  }
}
```

DynamoDB テーブルをすばやく確認すると、スキャンしたときにテーブルにエントリが表示されます。

id *(String)*	date	title
9f62c4dd-49d5-48d5-b835-143284c72fe0	1970-01-01T12:30:00.000Z	first post

AWS AppSync コンソールに戻り、クエリを実行してこの を取得するとPost、次の結果が得られます。

```
{
  "data": {
    "getPost": [
      {
        "id": "9f62c4dd-49d5-48d5-b835-143284c72fe0",
        "date": "1970-01-01T12:30:00.000Z",
```

```
        "title": "first post"
      }
    ]
  }
}
```

でのリアルタイムデータアプリケーションのサブスクリプションの使用 AWS AppSync

> **Important**
>
> 2025 年 3 月 13 日以降、 AWS AppSync Events を使用して WebSockets を搭載したリアルタイム PubSub API を構築できます。詳細については、AppSync Events デベロッパーガイドのWebSocket 経由でイベントを発行する」を参照してください。 AWS AppSync

AWS AppSync では、サブスクリプションを利用して、ライブアプリケーションの更新、プッシュ通知などを実装できます。クライアントが GraphQL サブスクリプションオペレーションを呼び出すと、 AWS AppSync によって安全な WebSocket 接続が自動的に確立され、維持されます。その後、アプリケーションはデータソースからサブスクライバーにデータをリアルタイムで配信でき、 AWS AppSync はアプリケーションの接続とスケーリング要件を継続的に管理します。以下のセクションでは、 AWS AppSync のサブスクリプションの仕組みを示します。

GraphQL スキーマサブスクリプションディレクティブ

AWS AppSync のサブスクリプションは、ミューテーションへの応答として呼び出されます。つまり、ミューテーションに GraphQL スキーマディレクティブを指定することで、 AWS AppSync のデータソースをリアルタイムで作成できます。

AWS Amplify クライアントライブラリは、サブスクリプション接続管理を自動的に処理します。ライブラリは純粋な WebSockets をクライアントとサービス間のネットワークプロトコルとして使用します。

> **Note**
>
> サブスクリプションへの接続時に認可を制御するには、 AWS Identity and Access Management (IAM) AWS Lambda、、Amazon Cognito ID プール、または Amazon Cognito

ユーザープールを使用してフィールドレベルの認可を行うことができます。サブスクリプションのきめ細かなアクセスコントロールの場合、リゾルバーをサブスクリプションフィールドにアタッチし、発信者と AWS AppSync データソースの ID を使用してロジックを実行できます。詳細については、「GraphQL API を保護するための認可と認証の設定」を参照してください。

サブスクリプションはミューテーションからトリガーされ、ミューテーション選択セットが受信者に送信されます。

次の例では、GraphQL サブスクリプションの操作方法を示します。データソースを指定しません。これは、データソースが 、Amazon DynamoDB、または Amazon OpenSearch Service のいずれかであるからです。

サブスクリプションを開始するには、次のようにスキーマにサブスクリプションのエントリポイントを追加します。

```
schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
}
```

ブログ投稿サイトを持っており、新しいブログにサブスクライブし、既存のブログに変わるものとします。これを行うには、スキーマに次の Subscription 定義を追加します。

```
type Subscription {
    addedPost: Post
    updatedPost: Post
    deletedPost: Post
}
```

さらに、次のミューテーションがあるとします。

```
type Mutation {
    addPost(id: ID! author: String! title: String content: String url: String): Post!
    updatePost(id: ID! author: String! title: String content: String url: String ups:
 Int! downs: Int! expectedVersion: Int!): Post!
    deletePost(id: ID!): Post!
```

```
}
```

通知を受け取る各サブスクリプションに @aws_subscribe(mutations: ["mutation_field_1", "mutation_field_2"]) ディレクティブを追加することで、これらのフィールドをリアルタイム対応にできます。手順は以下のとおりです。

```
type Subscription {
    addedPost: Post
    @aws_subscribe(mutations: ["addPost"])
    updatedPost: Post
    @aws_subscribe(mutations: ["updatePost"])
    deletedPost: Post
    @aws_subscribe(mutations: ["deletePost"])
}
```

@aws_subscribe(mutations: ["",..,""]) は、ミューテーション入力の配列を受け取るため、サブスクリプションを開始する複数のミューテーションを指定できます。クライアントからサブスクライブする場合、GraphQL クエリは次のようになります。

```
subscription NewPostSub {
    addedPost {
        __typename
        version
        title
        content
        author
        url
    }
}
```

このサブスクリプションクエリは、クライアント接続とツールに必要です。

ピュアな WebSocket クライアントを使用する場合、各クライアントが独自の選択セットを定義できるため、選択セットのフィルタリングはクライアントごとに行われます。この場合、サブスクリプション選択セットは、ミューテーション選択セットのサブセットである必要があります。例えば、サブスクリプションが addedPost{author title} ミューテーションにリンクされていると、addPost(...){id author title url version} 投稿の作成者とタイトルのみを受け取ります。他のフィールドは受け取りません。ただし、ミューテーションの選択セットに作成者が欠けていた場合、サブスクライバーは作成者フィールドに対して null 値を取得します (または、スキーマ内で作成者フィールドが required/not-null と定義されている場合はエラー)。

ピュアな WebSockets を使用する場合は、サブスクリプション選択セットが不可欠です。サブスクリプションでフィールドが明示的に定義されていない場合、 AWS AppSync は フィールドを返しません。

前の例では、サブスクリプションに引数はありませんでした。次のようなスキーマがあるとします。

```
type Subscription {
    updatedPost(id:ID! author:String): Post
    @aws_subscribe(mutations: ["updatePost"])
}
```

この場合、クライアントでサブスクリプションが次のように定義されます。

```
subscription UpdatedPostSub {
    updatedPost(id:"XYZ", author:"ABC") {
        title
        content
    }
}
```

スキーマの subscription フィールドの戻り値の型は、対応する mutation フィールドの戻り値の型に一致する必要があります。前の例では、これが addPost と addedPost の両方が Post タイプとして返され、表示されます。

クライアントでサブスクリプションをセットアップするには、「Amplify クライアントを使用したクライアントアプリケーションの構築」を参照してください。

サブスクリプション引数の使用

GraphQL サブスクリプションを使用する上で重要なのは、引数をいつ、どのように使うかを理解することです。微妙な変更を加えることで、発生したミューテーションをクライアントに通知する方法とタイミングを変更できます。そのためには、クイックスタートの章にある「Todos」を作成するサンプルスキーマを参照してください。このサンプルスキーマでは、以下のミューテーションが定義されます。

```
type Mutation {
    createTodo(input: CreateTodoInput!): Todo
    updateTodo(input: UpdateTodoInput!): Todo
    deleteTodo(input: DeleteTodoInput!): Todo
}
```

デフォルトのサンプルでは、クライアントは引数なしで onUpdateTodo subscription を使うことで、任意の Todo への更新を購読できます。

```
subscription OnUpdateTodo {
  onUpdateTodo {
    description
    id
    name
    when
  }
}
```

引数を使用して subscription をフィルタリングできます。例えば、特定の ID を含む todo が更新された場合に subscription のみをトリガーするには、ID 値を指定します。

```
subscription OnUpdateTodo {
  onUpdateTodo(id: "a-todo-id") {
    description
    id
    name
    when
  }
}
```

複数の引数を渡すこともできます。例えば、次の subscription は、特定の場所と時間に Todo 更新があった場合に通知を受ける方法を示しています。

```
subscription todosAtHome {
  onUpdateTodo(when: "tomorrow", where: "at home") {
    description
    id
    name
    when
    where
  }
}
```

引数はすべてオプションであることに注意してください。subscription で引数を指定しない場合、アプリケーションで発生するすべての Todo 更新を購読することになります。ただし、subscription のフィールド定義を更新して ID 引数を必須にすることはできます。これにより、すべての todo の代わりに、特定の todo への応答が強制されます。

```
onUpdateTodo(
  id: ID!,
  name: String,
  when: String,
  where: String,
  description: String
): Todo
```

引数値 null には意味があります。

AWS AppSync でサブスクリプションクエリを作成する場合、null引数の値は引数を完全に省略するのとは異なる方法で結果をフィルタリングします。

todo を作成できる todos API サンプルに戻りましょう。クイックスタートの章にあるサンプルスキーマを参照してください。

スキーマを変更して、所有者が誰であるかを記述する新しい owner フィールドを Todo タイプに追加してみましょう。owner フィールドは必須ではなく、UpdateTodoInput に設定することしかできません。スキーマの次の簡略版を参照してください。

```
type Todo {
  id: ID!
  name: String!
  when: String!
  where: String!
  description: String!
  owner: String
}

input CreateTodoInput {
  name: String!
  when: String!
  where: String!
  description: String!
}

input UpdateTodoInput {
  id: ID!
  name: String
  when: String
  where: String
  description: String
```

```
  owner: String
}

type Subscription {
    onUpdateTodo(
        id: ID,
        name: String,
        when: String,
        where: String,
        description: String
    ): Todo @aws_subscribe(mutations: ["updateTodo"])
}
```

次のサブスクリプションはすべての Todo 更新を返します。

```
subscription MySubscription {
  onUpdateTodo {
    description
    id
    name
    when
    where
  }
}
```

前のサブスクリプションを変更してフィールド引数 owner: null を追加すると、別の質問をすることになります。これで、所有者が指定されていないすべての Todo 更新について通知を受けることができるように、このサブスクリプションによってクライアントが登録されるようになりました。

```
subscription MySubscription {
  onUpdateTodo(owner: null) {
    description
    id
    name
    when
    where
  }
}
```

> **Note**
>
> 2022 年 1 月 1 日現在、MQTT over WebSockets は API の GraphQL サブスクリプションのプロトコルとして利用できなくなりました。 AWS AppSync APIs Pure WebSockets は、 でサポートされている唯一のプロトコルです AWS AppSync。
> 2019 年 11 月以降にリリースされた AWS AppSync SDK または Amplify ライブラリに基づくクライアントは、デフォルトで純粋な WebSocketsを自動的に使用します。クライアントを最新バージョンにアップグレードすると、 AWS AppSyncクライアントは純粋な WebSockets エンジンを使用できます。
> 純粋な WebSockets には、より大きなペイロードサイズ (240KB)、幅広いクライアントオプション、改善された CloudWatch メトリックスが付属しています。純粋な WebSocket クライアントを使用する方法の詳細については、「AWS AppSync でのリアルタイム WebSocket クライアントの構築」を参照してください。

でのサーバーレス WebSockets を使用した汎用 pub/sub APIs の作成 AWS AppSync

> **Important**
>
> 2025 年 3 月 13 日以降、 AWS AppSync Events を使用して WebSockets を搭載したリアルタイム PubSub API を構築できます。詳細については、AppSync Events デベロッパーガイドのWebSocket 経由でイベントを発行する」を参照してください。 AWS AppSync

一部のアプリケーションでは、クライアントが特定のチャネルまたはトピックをリスンする単純な WebSocket API のみが必要です。特定の形式や厳密に型指定された要件のない汎用 JSON データを、純粋でシンプルなパブリッシュ/サブスクライブ (pub/sub) パターンで、これらのチャネルのいずれかをリスニングしているクライアントにプッシュできます。

AWS AppSync を使用して、APIs バックエンドとクライアント側の両方で GraphQL コードを自動的に生成することで、GraphQL の知識がほとんどまたはまったくないシンプルな pub/sub WebSocket API を数分で実装します。

Pub/sub API を作成して設定する

開始するには、次のことを行います。

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   - ダッシュボードで、[API の作成] を選択します。
2. 次の画面で [リアルタイム API の作成] を選択し、[次へ] を選択します。
3. Pub/sub API のわかりやすい名前を入力します。
4. プライベート API 機能を有効にすることもできますが、現時点ではオフにしておくことをおすすめします。[次へ]を選択します。
5. WebSockets を使用して、動作する Pub/sub API を自動的に生成するように選択できます。今のところこの機能はオフにしておくことをおすすめします。[次へ]を選択します。
6. [API を作成] を選択し、数分待ってください。アカウントに新しい事前設定された AWS AppSync pub/sub API が作成されます AWS 。

API は、 AWS AppSync の組み込みローカルリゾルバー (ローカルリゾルバーの使用の詳細については、AWS AppSync デベロッパーガイド」の「チュートリアル: ローカルリゾルバー」を参照) を使用して、複数の一時 pub/sub チャネルと WebSocket 接続を管理します。これにより、チャネル名のみに基づいてサブスクライブされたクライアントに自動的にデータが配信およびフィルタリングされます。API コールは API キーを使用して承認されます。

API がデプロイされると、クライアントコードを生成してクライアントアプリケーションと統合するための追加手順がいくつか表示されます。クライアントを素早く統合する方法の例として、このガイドではシンプルな React Web アプリケーションを使用します。

1. まず、ローカルマシンで NPM を使って定型的な React アプリを作成します。

```
$ npx create-react-app mypubsub-app
$ cd mypubsub-app
```

> **Note**
>
> この例では、Amplify ライブラリを使用してクライアントをバックエンド API に接続します。ただし、Amplify CLI プロジェクトをローカルで作成する必要はありません。この例では React が最適なクライアントですが、Amplify ライブラリは iOS、Android、Flutter クライアントもサポートしており、これらの異なるランタイムでも同じ機能を提供します。サポートされている Amplify クライアントは、 AWS AppSync GraphQL API バックエンドとやり取りするためのシンプルな抽象化を提供します。これには、AppSync リアルタイム WebSocket プロトコルと完全に互換性がある組み込み WebSocket 機能が含まれます。 AWS AppSync WebSocket

```
$ npm install @aws-amplify/api
```

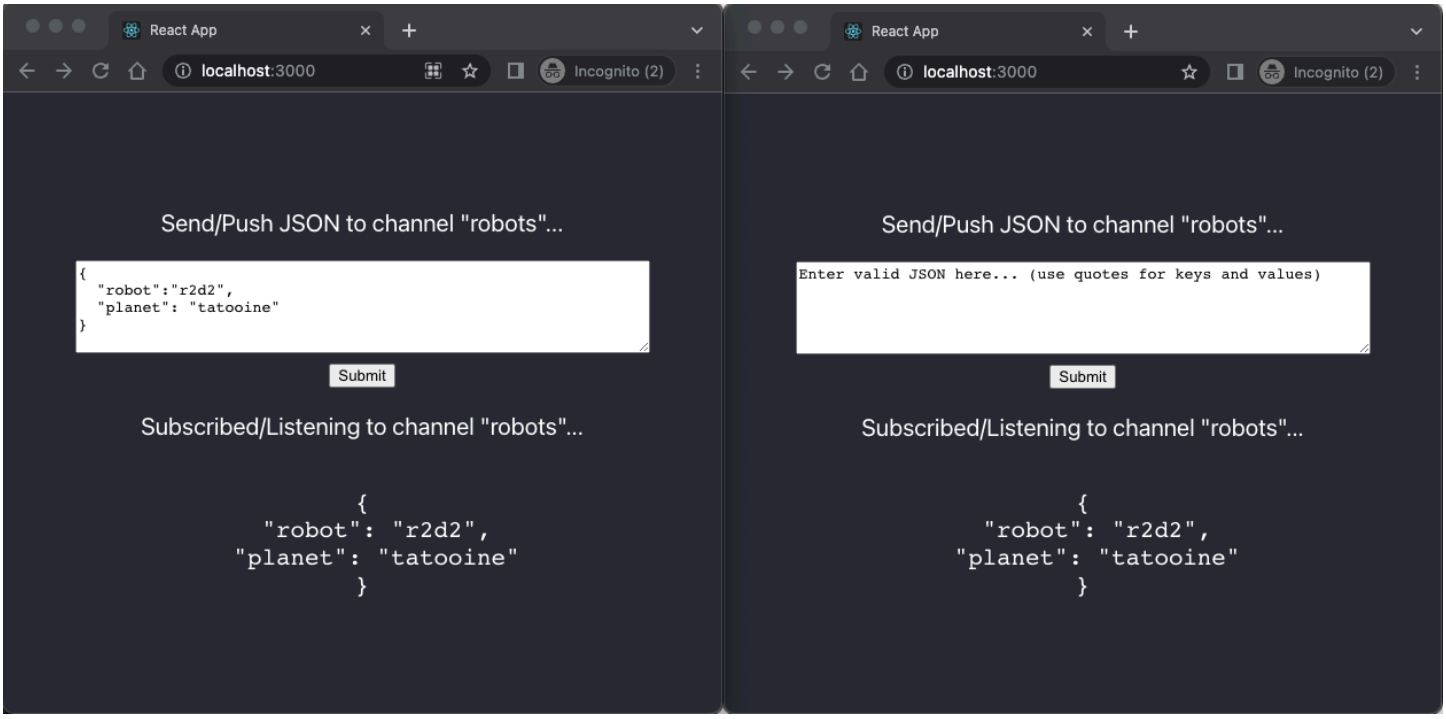
2. AWS AppSync コンソールで JavaScript を選択し、ダウンロードして API 設定の詳細と生成された GraphQL オペレーションコードを含む単一のファイルをダウンロードします。
3. ダウンロードしたファイルを React プロジェクトの `/src` フォルダーにコピーします。
4. 次に、既存のボイラープレート `src/App.js` ファイルの内容を、コンソールにあるサンプルクライアントコードに置き換えます。
5. 以下のコマンドを使用して、アプリケーションをローカルで構築し起動します。

```
$ npm start
```

6. リアルタイムデータの送受信をテストするには、2 つのブラウザーウィンドウを開いて *`localhost:3000`* にアクセスします。サンプルアプリケーションは、一般的な JSON データを *`robots`* という名前のハードコードされたチャネルに送信するように設定されています。
7. ブラウザウィンドウの 1 つで、テキストボックスに次の JSON blob を入力し、[送信] をクリックします。

```
{
  "robot":"r2d2",
  "planet": "tatooine"
}
```

どちらのブラウザーインスタンスも *`robots`* チャンネルに登録され、ウェブアプリケーションの下部に表示される公開データをリアルタイムで受信します。



スキーマ、リゾルバー、オペレーションなど、必要なすべての GraphQL API コードが自動的に生成され、汎用的な pub/sub ユースケースが可能になります。バックエンドでは、次のような GraphQL ミューテーションを使用してデータが AWS AppSync のリアルタイムエンドポイントに発行されます。

```
mutation PublishData {
    publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
        data
        name
    }
}
```

サブスクライバーは、関連する GraphQL サブスクリプションを使用して特定の一時チャンネルに送信された公開データにアクセスします。

```
subscription SubscribeToData {
    subscribe(name:"channel") {
        name
        data
    }
}
```

既存のアプリケーションへの pub-sub API の実装

既存のアプリケーションにリアルタイム機能を実装するだけであれば、この汎用的な pub/sub API 構成をあらゆるアプリケーションや API テクノロジーに簡単に統合できます。単一の API エンドポイントを使用して、GraphQL を使用した 1 回のネットワーク呼び出しで 1 つ以上のデータソースのデータに安全にアクセスし、操作し、結合する利点がありますが、 AWS AppSync のリアルタイム機能を活用するために既存の REST ベースのアプリケーションをゼロから変換または再構築する必要はありません。例えば、既存の CRUD ワークロードを別の API エンドポイントに配置し、クライアントは既存のアプリケーションから汎用 pub/sub API に対してリアルタイムおよび pub/sub のみを目的としてメッセージまたはイベントを送受信できます。

での拡張サブスクリプションフィルターの定義 AWS AppSync

> **Important**
>
> 2025 年 3 月 13 日以降、 AWS AppSync Events を使用して WebSockets を搭載したリアルタイム PubSub API を構築できます。詳細については、AppSync Events デベロッパーガイドのWebSocket 経由でイベントを発行する」を参照してください。 AWS AppSync

では AWS AppSync、追加の論理演算子をサポートするフィルターを使用して、GraphQL API サブスクリプションリゾルバーでバックエンドのデータフィルタリングのビジネスロジックを直接定義して有効にできます。クライアントのサブスクリプションクエリで定義されるサブスクリプション引数とは異なり、これらのフィルターは設定できます。サブスクリプション引数の使用の詳細については、「サブスクリプション引数の使用」を参照してください。演算子のリストについては、「AWS AppSync リゾルバーマッピングテンプレートユーティリティリファレンス」を参照してください。

このドキュメントでは、リアルタイムデータフィルタリングを次のカテゴリに分類します。

- 基本フィルタリング - サブスクリプションクエリのクライアント定義引数に基づくフィルタリング。
- 拡張フィルタリング - AWS AppSync サービスバックエンドで一元的に定義されたロジックに基づくフィルタリング。

以下のセクションでは、拡張サブスクリプションフィルターの設定方法と、その実際の使用方法について説明します。

GraphQL スキーマでのサブスクリプションの定義

拡張サブスクリプションフィルターを使用するには、GraphQL スキーマでサブスクリプションを定義し、次にフィルター拡張機能を使用して拡張フィルターを定義します。拡張サブスクリプションフィルタリングの仕組みを説明するには AWS AppSync、チケット管理システム API を定義する次の GraphQL スキーマを例として使用します。

```
type Ticket {
 id: ID
 createdAt: AWSDateTime
 content: String
 severity: Int
 priority: Priority
 category: String
 group: String
 status: String

}

type Mutation {
 createTicket(input: TicketInput): Ticket
}

type Query {
 getTicket(id: ID!): Ticket
}

type Subscription {
 onSpecialTicketCreated: Ticket @aws_subscribe(mutations: ["createTicket"])
 onGroupTicketCreated(group: String!): Ticket @aws_subscribe(mutations:
 ["createTicket"])
}



enum Priority {
 none
 lowest
 low
 medium
 high
 highest
}
```

```
input TicketInput {
 content: String
 severity: Int
 priority: Priority
 category: String
 group: String
```

API NONE のデータソースを作成し、このデータソースを使用して createTicket ミューテーションにリゾルバーをアタッチするとします。ハンドラーは次のようになります。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
 return {
  payload: {
   id: util.autoId(),
   createdAt: util.time.nowISO8601(),
   status: 'pending',
   ...ctx.args.input,
  },
 };
}

export function response(ctx) {
 return ctx.result;
}
```

> **Note**
>
> 拡張フィルターは、特定のサブスクリプションの GraphQL リゾルバーのハンドラーで有効になります。詳細については、「リゾルバーのリファレンス」を参照してください。

拡張フィルターの動作を実装するには、extensions.setSubscriptionFilter() 関数を使用して、サブスクライブされたクライアントが関心を持つ可能性のある GraphQL ミューテーションからの公開データに対して評価されるフィルター式を定義する必要があります。 拡張機能の詳細については、「拡張機能」を参照してください。

次のセクションでは、フィルターを使用して、拡張フィルターを使用する方法について説明します。

フィルター拡張機能を使用して拡張サブスクリプションフィルターを作成する

拡張フィルターは、サブスクリプションのリゾルバーのレスポンスハンドラーに JSON で記述されます。フィルターは filterGroup というリストにまとめることができます。フィルターは、それぞれフィールド、演算子、値を含む少なくとも 1 つのルールを使用して定義されます。拡張フィルターを設定するための onSpecialTicketCreated 向けの新しいリゾルバーを定義しましょう。フィルターには AND ロジックを使用して評価される複数のルールを設定でき、フィルターグループ内の複数のフィルターは OR ロジックを使用して評価されます。

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
 // simplfy return null for the payload
 return { payload: null };
}

export function response(ctx) {
 const filter = {
  or: [
   { severity: { ge: 7 }, priority: { in: ['high', 'medium'] } },
   { category: { eq: 'security' }, group: { in: ['admin', 'operators'] } },
  ],
 };
 extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

  // important: return null in the response
 return null;
}
```

前の例で定義したフィルターに基づいて、チケットが以下のように作成された場合、重要なチケットはサブスクライブされている API クライアントに自動的にプッシュされます。

- priority レベル high または medium

  AND

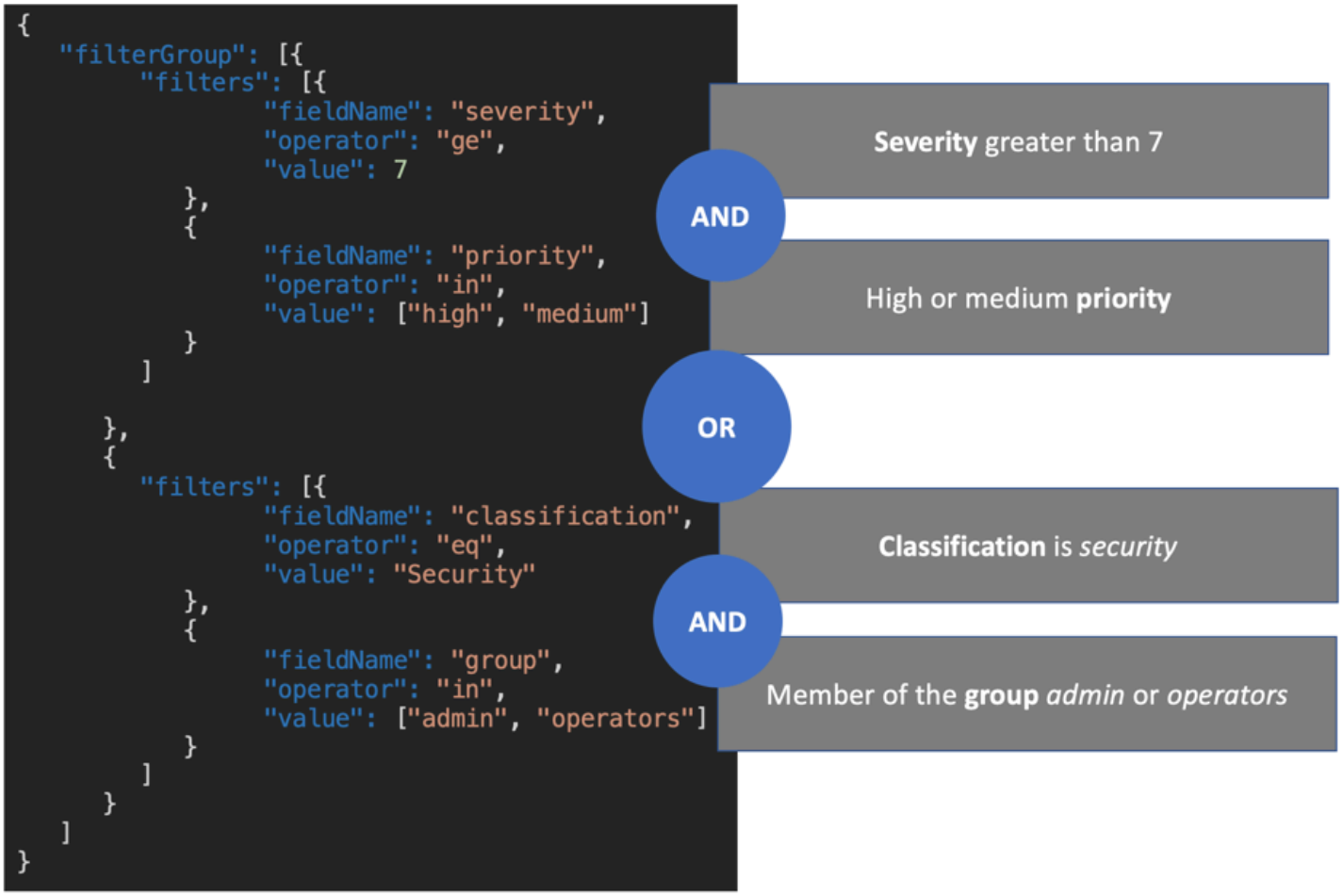
- severity レベル 7より大きいか等しい (ge}

または

- classification チケットをSecurityに設定

AND

- groupアサインをadminまたはoperatorsに設定



サブスクリプションリゾルバーで定義されたフィルター (拡張フィルター) は、サブスクリプション引数のみに基づくフィルター (基本フィルター) よりも優先されます。サブスクリプション引数の使用について詳しくは、「サブスクリプション引数の使用」を参照してください。

サブスクリプションの GraphQL スキーマで引数が定義されていて必要な場合、その引数がリゾルバーの `extensions.setSubscriptionFilter()` メソッドでルールとして定義されている場合にのみ、指定された引数に基づくフィルタリングが行われます。ただし、サブスクリプションリゾルバーに `extensions` フィルタリングメソッドがない場合、クライアントで定義された引数は基本的なフィルタリングにのみ使用されます。基本フィルターと拡張フィルターは同時に使用できません。

サブスクリプションのフィルター拡張ロジック内の`context` 変数を使用して、リクエストに関するコンテキスト情報にアクセスできます。例えば、Amazon Cognito ユーザープール、OIDC、または

Lambda カスタムオーソライザーを認可に使用する場合、サブスクリプションが確立されたときに context.identity でユーザーに関する情報を取得できます。この情報を使用して、ユーザーの ID に基づいてフィルタを設定できます。

次に、拡張されたフィルタ動作を onGroupTicketCreated に実装すると仮定します。onGroupTicketCreated サブスクリプションには引数として必須の group 名が必要です。チケットを作成すると、自動的に pending ステータスが割り当てられます。サブスクリプションフィルターを設定して、指定したグループに属する、新しく作成されたチケットのみを受信できます。

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
 // simplfy return null for the payload
 return { payload: null };
}

export function response(ctx) {
 const filter = { group: { eq: ctx.args.group }, status: { eq: 'pending' } };
 extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

 return null;
}
```

以下の例のようにミューテーションを使用してデータが公開された場合:

```
mutation CreateTicket {
  createTicket(input: {priority: medium, severity: 2, group: "aws"}) {
    id
    priority
    severity
    status
    group
    createdAt
  }
}
```

サブスクライブされたクライアントは、createTicket ミューテーションを含むチケットが作成されるとすぐに、データが WebSockets を介して自動的にプッシュされるのを待ち受けます。

```
subscription OnGroup {
  onGroupTicketCreated(group: "aws") {
```

```
    category
    status
    severity
    priority
    id
    group
    createdAt
    content
  }
}
```

フィルタリングロジックが拡張フィルタリングで AWS AppSync サービスに実装されているため、クライアントは引数なしでサブスクライブできるため、クライアントコードが簡素化されます。クライアントは、定義されたフィルター条件が満たされた場合にのみデータを受け取ります。

ネストされたスキーマフィールド用の拡張フィルターの定義

強化されたサブスクリプションフィルターを使用して、ネストされたスキーマフィールドをフィルターできます。前のセクションのスキーマを口ケーションと住所のタイプを含むように変更したとします。

```
type Ticket {
 id: ID
 createdAt: AWSDateTime
 content: String
 severity: Int
 priority: Priority
 category: String
 group: String
 status: String
 location: ProblemLocation
}

type Mutation {
 createTicket(input: TicketInput): Ticket
}

type Query {
 getTicket(id: ID!): Ticket
}

type Subscription {
 onSpecialTicketCreated: Ticket @aws_subscribe(mutations: ["createTicket"])
```

```
 onGroupTicketCreated(group: String!): Ticket @aws_subscribe(mutations:
 ["createTicket"])
}

type ProblemLocation {
 address: Address
}

type Address {
 country: String
}

enum Priority {
 none
 lowest
 low
 medium
 high
 highest
}

input TicketInput {
 content: String
 severity: Int
 priority: Priority
 category: String
 group: String
 location: AWSJSON
```

このスキーマでは、. セパレータを使用してネストを表すことができます。次の例では、location.address.country の下にネストされたスキーマフィールドのフィルタルールを追加しています。チケットのアドレスが USA に設定されていると、サブスクリプションがトリガーされます。

```
import { util, extensions } from '@aws-appsync/utils';

export const request = (ctx) => ({ payload: null });

export function response(ctx) {
 const filter = {
  or: [
   { severity: { ge: 7 }, priority: { in: ['high', 'medium'] } },
   { category: { eq: 'security' }, group: { in: ['admin', 'operators'] } },
```

```
    { 'location.address.country': { eq: 'USA' } },
   ],
  };
  extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));
  return null;
}
```

上の例では、location がネストレベル 1、address がネストレベル 2、country がネストレベル 3 を表しており、これらはすべて . セパレータで区切られています。

このサブスクリプションは、createTicket ミューテーションを使用してテストできます。

```
mutation CreateTicketInUSA {
  createTicket(input: {location: "{\"address\":{\"country\":\"USA\"}}"}) {
    category
    content
    createdAt
    group
    id
    location {
      address {
        country
      }
    }
    priority
    severity
    status
  }
}
```

クライアントからの拡張フィルターの定義

GraphQL では、サブスクリプション引数を使用して基本的なフィルタリングを使用できます。サブスクリプションクエリで呼び出しを行うクライアントは、引数の値を定義します。extensions フィルターを使用して AWS AppSync サブスクリプションリゾルバーで拡張フィルターを有効にすると、リゾルバーで定義されたバックエンドフィルターが優先され、優先度が優先されます。

サブスクリプションの filter 引数を使用して、クライアント定義の動的な拡張フィルターを設定します。これらのフィルターを設定するときは、新しい引数を反映するように GraphQL スキーマを更新する必要があります。

```
...
```

```
type Subscription {
    onSpecialTicketCreated(filter: String): Ticket
        @aws_subscribe(mutations: ["createTicket"])
}
...
```

その後、クライアントは次の例のようにサブスクリプションクエリを送信できます。

```
subscription onSpecialTicketCreated($filter: String) {
     onSpecialTicketCreated(filter: $filter) {
        id
        group
        description
        priority
        severity
     }
 }
```

クエリ変数は、次の例のように設定できます。

```
{"filter" : "{\"severity\":{\"le\":2}}"}
```

`util.transform.toSubscriptionFilter()` リゾルバーユーティリティをサブスクリプション応答マッピングテンプレートに実装すると、サブスクリプション引数で定義されたフィルターを各クライアントに適用できます。

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
 // simplfy return null for the payload
 return { payload: null };
}

export function response(ctx) {
 const filter = ctx.args.filter;
 extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));
 return null;
}
```

この戦略により、クライアントは拡張フィルタリングロジックと追加の演算子を使用する独自のフィルターを定義できます。フィルターは、特定のクライアントがセキュア WebSocket 接続でサブスク

リプションクエリを呼び出すときに割り当てられます。filter クエリ変数ペイロードの形式など、拡張フィルタリング用の変換ユーティリティの詳細については、「JavaScript リゾルバーの概要」を参照してください。

その他の強化されたフィルタリング制限

拡張フィルターに追加の制限が課されるユースケースを以下に示します。

- 拡張フィルタは、最上位のオブジェクトリストのフィルタリングをサポートしていません。このユースケースでは、ミューテーションから公開されたデータは拡張サブスクリプションでは無視されます。
- AWS AppSync は最大 5 レベルのネストをサポートしています。ネストレベル 5 を超えるスキーマフィールドのフィルターは無視されます。以下の GraphQL レスポンスを見てください。venue.address.country.metadata.continent の continent フィールドはレベル 5 のネストなので許可されています。ただし、venue.address.country.metadata.capital.financial の financial はレベル 6 のネストなので、フィルターは機能しません。

```
{
    "data": {
        "onCreateFilterEvent": {
            "venue": {
                "address": {
                    "country": {
                        "metadata": {
                            "capital": {
                                "financial": "New York"
                            },
                            "continent" : "North America"
                        }
                    },
                    "state": "WA"
                },
                "builtYear": 2023
            },
            "private": false,
        }
    }
}
```

のフィルターを使用した WebSocket 接続のサブスクリプション解除 AWS AppSync

> **Important**
>
> 2025 年 3 月 13 日以降、 AWS AppSync Events を使用して WebSockets を搭載したリアルタイム PubSub API を構築できます。詳細については、AppSync Events デベロッパーガイドのWebSocket 経由でイベントを発行する」を参照してください。 AWS AppSync

では AWS AppSync、特定のフィルタリングロジックに基づいて、接続されたクライアントから WebSocket 接続を強制的にサブスクリプション解除および終了 (無効化) できます。これは、ユーザーをグループから削除する場合など、承認関連のシナリオで役立ちます。

サブスクリプションの無効化は、ミューテーションで定義されたペイロードに応じて行われます。サブスクリプション接続の無効化に使用されるミューテーションは API の管理操作として扱い、その使用を管理ユーザー、グループ、またはバックエンドサービスに限定して、それに応じて権限の範囲を設定することをおすすめします。例えば、`@aws_auth(cognito_groups: ["Administrators"])` または `@aws_iam` などのスキーマ認可ディレクティブを使用します。詳細については、「その他の認可モードを使用する」を参照してください。

無効化フィルターは、拡張サブスクリプションフィルターと同じ構文とロジックを使用します。これらのフィルターは、以下のユーティリティを使用して定義します。

- `extensions.invalidateSubscriptions()` – GraphQL リゾルバーのミューテーションのレスポンスハンドラーで定義されます。
- `extensions.setSubscriptionInvalidationFilter()` – ミューテーションにリンクされたサブスクリプションの GraphQL リゾルバーのレスポンスハンドラーで定義されます。

無効化フィルタリング拡張機能の詳細については、「JavaScript リゾルバーの概要」を参照してください。

サブスクリプションの無効化の使用

サブスクリプション無効化の仕組みを確認するには AWS AppSync、次の GraphQL スキーマを使用します。

```
type User {
```

```
  userId: ID!
  groupId: ID!
}

type Group {
  groupId: ID!
  name: String!
  members: [ID!]!
}

type GroupMessage {
  userId: ID!
  groupId: ID!
  message: String!
}

type Mutation {
    createGroupMessage(userId: ID!, groupId : ID!, message: String!): GroupMessage
    removeUserFromGroup(userId: ID!, groupId : ID!) : User @aws_iam
}

type Subscription {
    onGroupMessageCreated(userId: ID!, groupId : ID!): GroupMessage
        @aws_subscribe(mutations: ["createGroupMessage"])
}

type Query {
 none: String
}
```

removeUserFromGroup ミューテーションリゾルバーコードで無効化フィルターを定義します。

```
import { extensions } from '@aws-appsync/utils';

export function request(ctx) {
 return { payload: null };
}

export function response(ctx) {
 const { userId, groupId } = ctx.args;
 extensions.invalidateSubscriptions({
  subscriptionField: 'onGroupMessageCreated',
  payload: { userId, groupId },
```

```
 });
 return { userId, groupId };
}
```

ミューテーションが呼び出されると、payload オブジェクトに定義されているデータを使用して、subscriptionField で定義されたサブスクリプションの購読が解除されます。onGroupMessageCreated サスクリプションのレスポンスマッピングテンプレートには無効化フィルターも定義されています。

extensions.invalidateSubscriptions() ペイロードに、フィルターで定義されているサブスクライブされたクライアントの ID と一致する ID が含まれている場合、対応するサブスクリプションはサブスクライブ解除されます。さらに、WebSocket 接続は閉じられます。onGroupMessageCreated サブスクリプションのサブスクリプションリゾルバーコードを定義します。

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
 // simplfy return null for the payload
 return { payload: null };
}

export function response(ctx) {
 const filter = { groupId: { eq: ctx.args.groupId } };
 extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

 const invalidation = { groupId: { eq: ctx.args.groupId }, userId: { eq:
 ctx.args.userId } };
 extensions.setSubscriptionInvalidationFilter(util.transform.toSubscriptionFilter(invalidation)

 return null;
}
```

サブスクリプションレスポンスハンドラーには、サブスクリプションフィルターと無効化フィルターの両方を同時に定義できることに注意してください。

例えば、クライアント A が次のサブスクリプションリクエストを使用して、その ID *group-1* を持つ新しいユーザーを ID *user-1* のグループにサブスクライブするとします。

```
onGroupMessageCreated(userId : "user-1", groupId: :"group-1"){...}
```

AWS AppSync はサブスクリプションリゾルバーを実行します。これにより、前述のonGroupMessageCreatedレスポンスマッピングテンプレートで定義されているサブスクリプションフィルターと無効化フィルターが生成されます。クライアント A の場合、サブスクリプションフィルタはへのデータ送信のみを許可し*group-1*、無効化フィルタは *user-1* と *group-1* の両方に定義されます。

ここで、クライアント B が次のサブスクリプションリクエストを使用して、その ID *user-2* を持つユーザーをその ID *group-2*のグループにサブスクライブすると仮定します。

```
onGroupMessageCreated(userId : "user-2", groupId: :"group-2"){...}
```

AWS AppSync はサブスクリプションリゾルバーを実行し、サブスクリプションフィルターと無効化フィルターを生成します。クライアント B の場合、サブスクリプションフィルタはデータ送信先を *group-2* のみを許可し、無効化フィルタは *user-2* と *group-2* の両方に定義されます。

次に、ID *message-1*の付いた新しいグループメッセージが、次の例のようにミューテーションリクエストを使用して作成されると仮定します。

```
createGroupMessage(id: "message-1", groupId :
      "group-1", message: "test message"){...}
```

定義されたフィルターに一致するサブスクライブされたクライアントは、WebSockets を介して次のデータペイロードを自動的に受信します。

```
{
  "data": {
    "onGroupMessageCreated": {
      "id": "message-1",
      "groupId": "group-1",
      "message": "test message",
    }
  }
}
```

フィルター条件が定義済みのサブスクリプションフィルターと一致するため、クライアント A はメッセージを受信します。ただし、ユーザーは *group-1* に参加していないため、クライアント B はメッセージを受信しません。また、リクエストはサブスクリプションリゾルバーで定義されているサブスクリプションフィルターと一致しません。

最後に、*user-1*が次のミューテーションリクエストの使用して *group-1* から削除されたと仮定します。

```
removeUserFromGroup(userId: "user-1", groupId : "group-1"){...}
```

ミューテーションは、extensions.invalidateSubscriptions()リゾルバーレスポンスハンドラーコードで定義されているサブスクリプションの無効化を開始します。 AWS AppSync その後、クライアント A のサブスクリプションを解除し、WebSocket 接続を閉じます。ミューテーションで定義された無効化ペイロードがユーザーまたはグループと一致しないため、クライアント B は影響を受けません。

が接続 AWS AppSync を無効にすると、クライアントはサブスクライブ解除されたことを確認するメッセージを受け取ります。

```
{
  "message": "Subscription complete."
}
```

サブスクリプション無効化フィルターでのコンテキスト変数の使用

拡張サブスクリプションフィルターと同様に、サブスクリプション無効化フィルター拡張の[context変数]を使用して特定のデータにアクセスできます。

例えば、ミューテーションの無効化ペイロードとして E メールアドレスを設定し、そのアドレスを Amazon Cognito ユーザープールまたは OpenID Connect で認証されたサブスクライブユーザーからのメール属性またはクレームと照合することができます。extensions.setSubscriptionInvalidationFilter() サブスクリプション無効化ツールで定義された無効化フィルターは、ミューテーションの extensions.invalidateSubscriptions() ペイロードによって設定されたメールアドレスが、context.identity.claims.email でユーザーの JWT トークンから取得したメールアドレスと一致するかどうかを確認し、無効化を開始します。

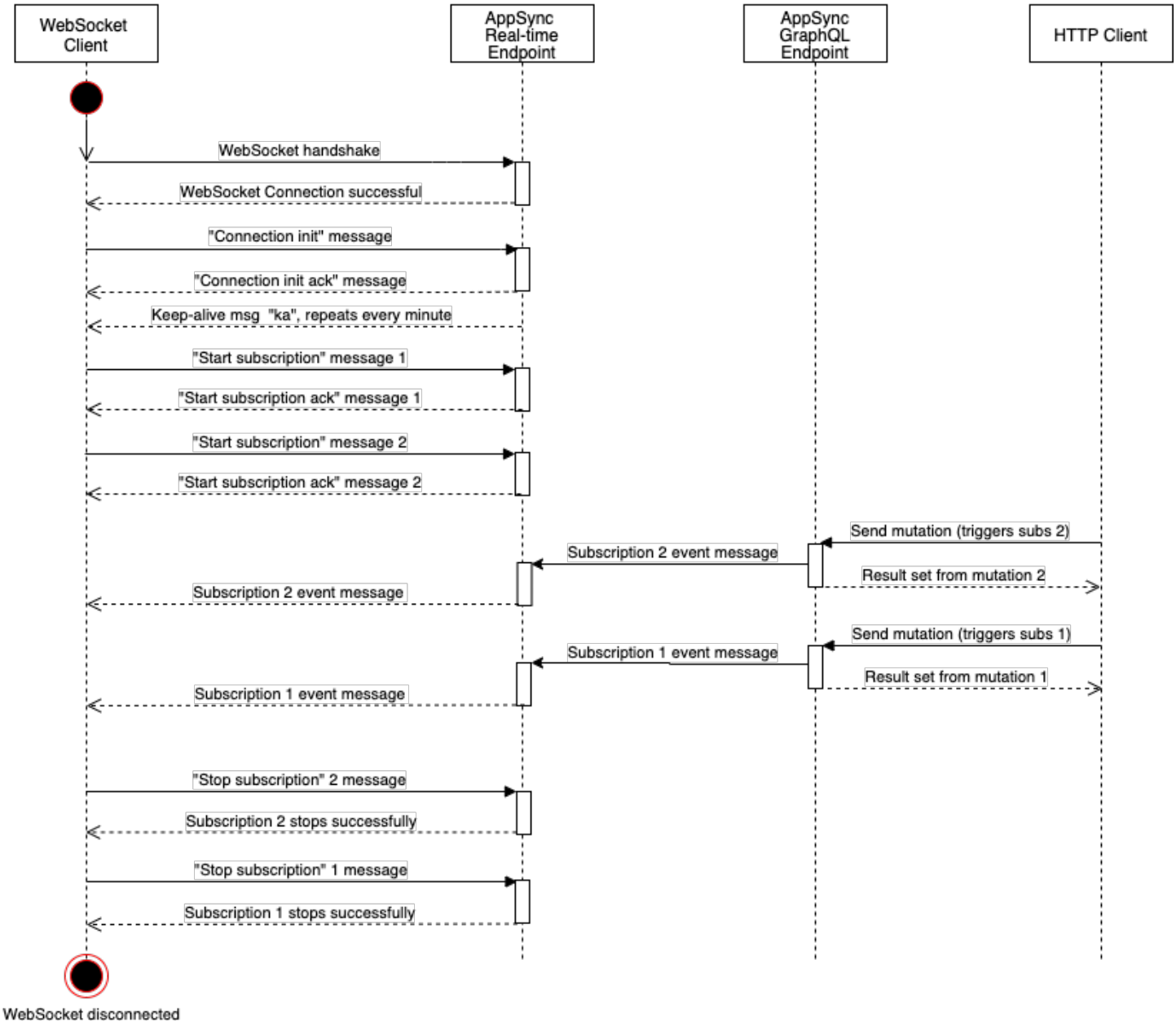
AWS AppSync でのリアルタイム WebSocket クライアントの構築

> **Important**
>
> 2025 年 3 月 13 日以降、 AWS AppSync Events を使用して WebSockets を搭載したリアルタイム PubSub API を構築できます。詳細については、AppSync Events デベロッパーガイドのWebSocket 経由でイベントを発行する」を参照してください。 AWS AppSync

AWS AppSync のリアルタイム WebSocket クライアントは、複数ステップのプロセスを通じて GraphQL サブスクリプションを有効にします。クライアントは、まず AWS AppSync リアルタイムエンドポイントとの WebSocket 接続を確立し、接続初期化メッセージを送信し、確認を待機します。接続が成功すると、クライアントは一意の IDsと GraphQL クエリで開始メッセージを送信してサブスクリプションを登録します。 AWS AppSync は、成功したサブスクリプションを承認メッセージで確認します。クライアントは、対応するミューテーションの呼び出しによってトリガーされるサブスクリプションイベントを監視します。接続を維持するために、 AWS AppSync は定期的なキープアライブメッセージを送信します。終了すると、クライアントは停止メッセージを送信してサブスクリプションを登録解除します。このシステムは、単一の WebSocket 接続で複数のサブスクリプションをサポートし、API キー、Amazon Cognito ユーザープール、IAM、Lambda など、さまざまな認可モードに対応します。

GraphQL サブスクリプションのリアルタイム WebSocket クライアント実装

次のシーケンス図とステップは、WebSocket クライアント、HTTP クライアント、 AWS AppSync 間のリアルタイムサブスクリプションワークフローを示しています。



1. クライアントは、 AWS AppSync リアルタイムエンドポイントとの WebSocket 接続を確立します。ネットワークエラーが発生した場合、クライアントはジッターされたエクスポネンシャルバックオフを行う必要があります。詳細については、 AWS 「 アーキテクチャブログ」の「エクスポネンシャルバックオフとジッター」を参照してください。
2. (オプション) WebSocket 接続が正常に確立されると、クライアントは`connection_init`メッセージを送信します。
3. が送信されると、クライアント`connection_init`は AWS AppSync から の`connection_ack`メッセージを待機します。このメッセージには、`"ka"` (キープアライブ)

メッセージの最大待機時間 (ミリ秒) である connectionTimeoutMs パラメータが含まれています。

4. AWS AppSync は定期的に"ka"メッセージを送信します。クライアントは、各 "ka" メッセージを受信した時間を追跡します。クライアントが connectionTimeoutMs ミリ秒以内に "ka" メッセージを受信しない場合、クライアントは接続を終了する必要があります。
5. クライアントは、start サブスクリプションメッセージを送信してサブスクリプションを登録します。単一の WebSocket 接続では、異なる認可モードであっても、複数のサブスクリプションがサポートされます。
6. クライアントは、 AWS AppSync が成功したサブスクリプションを確認するためにstart_ackメッセージを送信するのを待ちます。エラーが発生した場合、 AWS AppSync は"type": "error"メッセージを返します。
7. クライアントは、対応するミューテーションが呼び出された後に送信されるサブスクリプションイベントを監視します。クエリとミューテーションは通常、 AWS AppSync GraphQL エンドポイントhttps://に送信されます。サブスクリプションは、安全な WebSocket () を使用して AWS AppSync リアルタイムエンドポイントを経由しますwss://。
8. クライアントは、stop サブスクリプションメッセージを送信してサブスクリプションを登録解除します。
9. すべてのサブスクリプションを登録解除した後、WebSocket を介して転送されるメッセージがないことを確認した後、クライアントは WebSocket 接続から切断できます。

WebSocket 接続を確立するためのハンドシェイクの詳細

AWS AppSync で正常なハンドシェイクに接続して開始するには、WebSocket クライアントに以下が必要です。

- AWS AppSync リアルタイムエンドポイント
- ヘッダー – AWS AppSync エンドポイントと認可に関連する情報が含まれています。 AWS AppSync は、ヘッダーを提供する次の 3 つの方法をサポートしています。
  - クエリ文字列によるヘッダー
    - ヘッダー情報は、文字列化された JSON オブジェクトから派生した base64 文字列として エンコードされます。この JSON オブジェクトには、 AWS AppSync エンドポイントと認可に関連する詳細が含まれています。JSON オブジェクトの内容は、認可モードによって異なります。
  - 経由のヘッダー Sec-WebSocket-Protocol

    - AWS AppSync エンドポイントと認可に関連する情報を含む文字列化された JSON オブジェクトからの base64Url-encodedされた文字列は、`Sec-WebSocket-Protocol`ヘッダーのプロトコルとして渡されます。JSON オブジェクトの内容は、認可モードによって異なります。
  - 標準 HTTP ヘッダー経由のヘッダー：
    - ヘッダーは、GraphQL クエリとミューテーションのヘッダーを AWS AppSync に渡す方法と同様に、接続リクエストで標準 HTTP ヘッダーとして渡すことができます。ただし、標準 HTTP ヘッダーを介してヘッダーを渡すことは、プライベート API 接続リクエストではサポートされていません。
- `payload` – の Base64-encodedされた文字列`payload`。ペイロードは、クエリ文字列を使用してヘッダーが指定されている場合にのみ必要です

これらの必要な詳細情報を使用して、WebSocket クライアントは、WebSocket プロトコルとして `graphql-ws` を使用して、クエリ文字列がある API リアルタイムエンドポイントを含む URL に接続できます。

GraphQL エンドポイントからの リアルタイムエンドポイントの検出

AWS AppSync GraphQL エンドポイントと AWS AppSync リアルタイムエンドポイントは、プロトコルとドメインがわずかに異なります。 (AWS Command Line Interface AWS CLI) コマンド を使用して GraphQL エンドポイントを取得できます`aws appsync get-graphql-api`。

AWS AppSync GraphQL エンドポイント：

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync リアルタイムエンドポイント：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/
graphql
```

アプリケーションは、クエリとミューテーションに任意の HTTP クライアントを使用して、 AWS AppSync GraphQL エンドポイント (`https://`) に接続できます。アプリケーションは、サブスクリプション用の任意の WebSocket クライアントを使用して、 AWS AppSync リアルタイムエンドポイント (`wss://`) に接続できます。

カスタムドメイン名を使用すると、1 つのドメインを使用して両方のエンドポイントを操作できます。例えば、カスタムドメインとして `api.example.com` 設定すると、次の URL を使用して GraphQL およびリアルタイムエンドポイントを操作できます。

AWS AppSync カスタムドメイン GraphQL エンドポイント：

```
https://api.example.com/graphql
```

AWS AppSync カスタムドメインのリアルタイムエンドポイント：

```
wss://api.example.com/graphql/realtime
```

AWS AppSync API 認可モードに基づくヘッダーパラメータフォーマット

接続クエリ文字列で使用されるheaderオブジェクトの形式は、 AWS AppSync API 認可モードによって異なります。オブジェクトの hostフィールドは AWS AppSync GraphQL エンドポイントを参照します。これは、リアルタイムエンドポイントに対してwss://呼び出しが行われた場合でも、接続を検証するために使用されます。ハンドシェイクを開始し、許可された接続を確立するには、payload が空の JSON オブジェクトである必要があります。ペイロードは、ヘッダーがクエリ文字列を介して渡される場合にのみ必要です。

以下のセクションでは、各認可モードのヘッダー形式を示します。

API キー

API キーヘッダー

ヘッダーの内容

- "host": <string>: AWS AppSync GraphQL エンドポイントまたはカスタムドメイン名のホスト。
- "x-api-key": <string>: AWS AppSync API 用に設定された API キー。

例

```
{
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com",
    "x-api-key":"da2-12345678901234567890123456"
}
```

クエリ文字列によるヘッダー

まず、 hostと を含む JSON オブジェクトx-api-keyが文字列に変換されます。次に、この文字列は base64 エンコードを使用してエンコードされます。結果の base64 エンコードされた文字列は、

AWS AppSync リアルタイムエンドポイントとの接続を確立するために header WebSocket URL にという名前のクエリパラメータとして追加されます。生成されるリクエスト URL の形式は次のとおりです。

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJob3N0IjoiZXhhbXBsZTEyMzQ1Njc4OTAwMDAuYXBwc3luYy1hcGkudXMtZWFzdC0xLmFtYXpvbmF3cy5jb20i
```

base64 でエンコードされたヘッダーオブジェクトに加えて、空の JSON オブジェクト {} も base64 でエンコードされ、WebSocket URL payloadに という名前の別のクエリパラメータとして含まれることに注意してください。

経由のヘッダー **Sec-WebSocket-Protocol**

host と を含む JSON オブジェクトx-api-keyは文字列に変換され、base64Url エンコードを使用してエンコードされます。結果の base64Url-encodedされた文字列には、 というプレフィックスが付けられますheader-。このプレフィックス付き文字列は、 AWS AppSync リアルタイムエンドポイントとの WebSocket 接続を確立するときに、 graphql-wsSec-WebSocket-Protocolヘッダーの に加えて新しいサブプロトコルとして使用されます。

生成されるリクエスト URL の形式は次のとおりです。

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

Sec-WebSocket-Protocol ヘッダーには次の値が含まれます。

```
"sec-websocket-protocol" : ["graphql-ws", "header-
ewogICAgImhvc3QiOiJleGFtcGxlMTIzNDU2Nzg5MDAwMC5hcHBzeW5jLWFwaS51cy1lYXN0LTEuYW1hem9uYXdzLmNvbSI
```

標準 HTTP ヘッダー経由のヘッダー

この方法では、 AWS AppSync リアルタイムエンドポイントとの WebSocket 接続を確立するときに、ホストと API キーの情報が標準の HTTP ヘッダーを使用して送信されます。生成されるリクエスト URL の形式は次のとおりです。

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

リクエストヘッダーには以下が含まれます。

```
"sec-websocket-protocol" : ["graphql-ws"]
"host":"example1234567890000.appsync-api.us-east-1.amazonaws.com",
```

```
"x-api-key":"da2-12345678901234567890123456"
```

Amazon Cognito ユーザープールと OpenID 接続 (OIDC)

Amazon Cognito と OIDCheader

ヘッダーの内容:

- `"Authorization": <string>`: JWT ID トークン。ヘッダーには Bearer スキームを使用できます。
- `"host": <string>`: AWS AppSync GraphQL エンドポイントまたはカスタムドメイン名のホスト。

例:

```
{

 "Authorization":"eyEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJHWEFLSXBieU5WNHhsQjEXAMPLEnM2WldvPSIsImFsZ
zEE2DJH7sHOl2zxYi7f-SmEGoh2AD8emxQRYajByz-rE4Jh0QOymN2Ys-ZIkMpVBTPgu-
TMWDyOHhDUmUj2OP82yeZ3wlZAtr_gM4LzjXUXmI_K2yGjuXfXTaa1mvQEBG0mQfVd7SfwXB-
jcv4RYVi6j25qgow9Ew52ufurPqaK-3WAKG32KpV8J4-Wejq8t0c-
yA7sb8EnB551b7TU93uKRiVVK3E55Nk5ADPoam_WYE45i3s5qVAP_-InW75NUoOCGTsS8YWMfb6ecHYJ-1j-
bzA27zaT9VjctXn9byNFZmEXAMPLExw",
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

クエリ文字列によるヘッダー

まず、 `host`と を含む JSON オブジェクト`Authorization`が文字列に変換されます。次に、この文字列は base64 エンコードを使用してエンコードされます。結果の base64 エンコードされた文字列は、 AWS AppSync リアルタイムエンドポイントとの接続を確立するために `header` WebSocket URL に という名前のクエリパラメータとして追加されます。生成されるリクエスト URL の形式は次のとおりです。

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

base64 でエンコードされたヘッダーオブジェクトに加えて、空の JSON オブジェクト {} も base64 でエンコードされ、WebSocket URL `payload`に という名前の別のクエリパラメータとして含まれることに注意してください。

経由のヘッダー **Sec-WebSocket-Protocol**

host と を含む JSON オブジェクトAuthorizationは文字列に変換され、base64Url エンコードを使用してエンコードされます。結果の base64Url-encodedされた文字列には、 というプレフィックスが付けられますheader-。このプレフィックス付き文字列は、 AWS AppSync リアルタイムエンドポイントとの WebSocket 接続を確立するときに、 graphql-wsSec-WebSocket-Protocolヘッダーの に加えて新しいサブプロトコルとして使用されます。

生成されるリクエスト URL の形式は次のとおりです。

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

Sec-WebSocket-Protocol ヘッダーには次の値が含まれます。

```
"sec-websocket-protocol" : ["graphql-ws", "header-
ewogICAgImhvc3QiOiJleGFtcGxlMTIzNDU2Nzg5MDAwMC5hcHBzeW5jLWFwaS51cy1lYXN0LTEuYW1hem9uYXdzLmNvbSI
```

標準 HTTP ヘッダー経由のヘッダー

この方法では、 AWS AppSync リアルタイムエンドポイントとの WebSocket 接続を確立するときに、ホストと認可の情報が標準の HTTP ヘッダーを使用して送信されます。生成されるリクエスト URL の形式は次のとおりです。

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

リクエストヘッダーには以下が含まれます。

```
"sec-websocket-protocol" : ["graphql-ws"]
"Authorization":"eyEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJHWEFLSXBieU5WNHhsQjEXAMPLEnM2WldvPSIsImFsZy
zEE2DJH7sHOl2zxYi7f-SmEGoh2AD8emxQRYajByz-rE4Jh0QOymN2Ys-ZIkMpVBTPgu-
TMWDyOHhDUmUj2OP82yeZ3wlZAtr_gM4LzjXUXmI_K2yGjuXfXTaa1mvQEBG0mQfVd7SfwXB-
jcv4RYVi6j25qgow9Ew52ufurPqaK-3WAKG32KpV8J4-Wejq8t0c-
yA7sb8EnB551b7TU93uKRiVVK3E55Nk5ADPoam_WYE45i3s5qVAP_-InW75NUoOCGTsS8YWMfb6ecHYJ-1j-
bzA27zaT9VjctXn9byNFZmEXAMPLExw",
"host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
```

IAM

IAM ヘッダー

ヘッダーの内容

- "accept": "application/json, text/javascript": 定数 <string> パラメータ。
- "content-encoding": "amz-1.0": 定数 <string> パラメータ。
- "content-type": "application/json; charset=UTF-8": 定数 <string> パラメータ。
- "host": <string>: これは AWS AppSync GraphQL エンドポイントのホストです。
  - "x-amz-date": <string>: タイムスタンプは UTC で、YYYYMMDD'T'HHMMSS'Z' の ISO 8601 形式である必要があります。たとえば、20150830T123600Z は有効なタイムスタンプです。タイムスタンプにミリ秒を含めないでください。詳細については、AWS 全般のリファレンス で「署名バージョン 4 で日付を扱う」を参照してください。
  - "X-Amz-Security-Token": <string>: AWS セッショントークン。一時的なセキュリティ認証情報を使用する場合に必要です。詳細については、IAM ユーザーガイドの「AWS リソースを使用した一時的なセキュリティ認証情報の使用」を参照してください。
  - "Authorization": <string>: AWS AppSync エンドポイント用の Signature Version 4 (SigV4) 署名情報。署名プロセスの詳細については、AWS 全般のリファレンス で「タスク 4: HTTP リクエストに署名を追加する」のタスク 4: HTTP を参照してください。

SigV4 署名の HTTP リクエストには、正規の URL が含まれています。これは AWS AppSync GraphQL エンドポイントに /connect が追加されたものです。サービスエンドポイント AWS リージョンは、 AWS AppSync API を使用しているリージョンと同じで、サービス名は「appsync」です。署名する HTTP リクエストは次のとおりです。

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql/
connect",
  data: "{}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

例

```
{
  "accept": "application/json, text/javascript",
  "content-encoding": "amz-1.0",
  "content-type": "application/json; charset=UTF-8",
```

```
  "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
  "x-amz-date": "20200401T001010Z",
  "X-Amz-Security-Token":
 "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXRoZWFEXAMPLEcwRQIgAh97Cljq7wOPL8KsxP3YtDuyc/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKEn0zyUhBEXAMPLEjj/////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUdAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtwR+9zF7NaMMMse07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZOzlZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnGlSNVx7SMcfovkFDqQamm
+88y1OwwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVYlj9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEfwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCkcCGFnwBNFLmfnbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpgqJsPKt582caFKArIFIvO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+tOLnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOKtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtKOOweDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA==",
  "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
 Signature=83EXAMPLEbcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}
```

クエリ文字列によるヘッダー

まず、 host (AWS AppSync GraphQL エンドポイント) と他の認可ヘッダーを含む JSON オブジェクトが文字列に変換されます。次に、この文字列は base64 エンコードを使用してエンコードされます。結果の base64 エンコードされた文字列は、 という名前のクエリパラメータとして WebSocket URL に追加されますheader。生成されるリクエスト URL の形式は次のとおりです。

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

base64 でエンコードされたヘッダーオブジェクトに加えて、空の JSON オブジェクト {} も base64 でエンコードされ、WebSocket URL payloadに という名前の別のクエリパラメータとして含まれることに注意してください。

経由のヘッダー **Sec-WebSocket-Protocol**

host およびその他の認可ヘッダーを含む JSON オブジェクトは文字列に変換され、base64Url エンコードを使用してエンコードされます。結果の base64Url-encodedされた文字列には、 というプレフィックスが付けられますheader-。このプレフィックス付き文字列は、 AWS AppSync リアルタイムエンドポイントとの WebSocket 接続を確立するときに、 Sec-WebSocket-Protocolヘッダーgraphql-wsの に加えて新しいサブプロトコルとして使用されます。

生成されるリクエスト URL の形式は次のとおりです。

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

Sec-WebSocket-Protocol ヘッダーには次の値が含まれます。

```
"sec-websocket-protocol" : ["graphql-ws", "header-
ew0KICAiYWNjZXB0IjogImFwcGxpY2F0aW9uL2pzb24sIHRleHQvamF2YXNjcmlwdCIsDQogICJjb250ZW50LWVuY29kaW5
```

標準 HTTP ヘッダー経由のヘッダー

この方法では、 AWS AppSync リアルタイムエンドポイントとの WebSocket 接続を確立するときに、ホストおよびその他の認可情報が標準の HTTP ヘッダーを使用して送信されます。生成されるリクエスト URL の形式は次のとおりです。

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

リクエストヘッダーには以下が含まれます。

```
"sec-websocket-protocol" : ["graphql-ws"]
"accept": "application/json, text/javascript",
"content-encoding": "amz-1.0",
"content-type": "application/json; charset=UTF-8",
"host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
"x-amz-date": "20200401T001010Z",
"X-Amz-Security-Token":
 "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXRoZWFEXAMPLEcwRQIgAh97Cljq7wOPL8KsxP3YtDuyc/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKEn0zyUhBEXAMPLEjj//////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUdAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtwR+9zF7NaMMMse07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZOzlZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnGlSNVx7SMcfovkFDqQamm
+88y1OwwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
```

```
+BwDg3ht2CrNvpwjVYlj9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEfwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCkcCGFnwBNFLmfnbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpgqJsPKt582caFKArIFIvO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+tOLnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOKtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtKOOweDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA==",
"Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
 Signature=83EXAMPLEbcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
```

カスタムドメインを使用してリクエストに署名するには

```
{
  url: "https://api.example.com/graphql/connect",
  data: "{}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

例

```
{
  "accept": "application/json, text/javascript",
  "content-encoding": "amz-1.0",
  "content-type": "application/json; charset=UTF-8",
  "host": "api.example.com",
  "x-amz-date": "20200401T001010Z",
  "X-Amz-Security-Token":
 "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXRoZWFEXAMPLEcwRQIgAh97Cljq7wOPL8KsxP3YtDuyc/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKEn0zyUhBEXAMPLEjj/////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUdAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtwR+9zF7NaMMMse07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZOzlZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnGlSNVx7SMcfovkFDqQamm
```

```
+88y1OwwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVYlj9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEfwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCkcCGFnwBNFLmfnbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpgqJsPKt582caFKArIFIvO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+tOLnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOKtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtKOOweDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA==",
  "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
 Signature=83EXAMPLEbcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}
```

クエリ文字列を含むリクエスト URL

```
wss://api.example.com/graphql?
header=eyEXAMPLEHQiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoE
```

> **Note**
>
> 1 つの WebSocket 接続には、(異なる認証モードであっても) 複数のサブスクリプションを設定できます。これを実装する 1 つの方法は、最初のサブスクリプション用の WebSocket 接続を作成し、最後のサブスクリプションが登録解除されたときにそれを閉じることです。最後のサブスクリプションが登録解除された直後にアプリケーションがサブスクライブされた場合に備えて、WebSocket 接続を閉じる前に数秒待つことで最適化できます。モバイルアプリの例では、ある画面から別の画面に変更するとき、アンマウントイベントでは、サブスクリプションを停止し、マウントイベントでは、別のサブスクリプションを開始します。

Lambda 認可

Lambda 認可ヘッダー

ヘッダーの内容

- `"Authorization": <string>`:`authorizationToken` として渡される値。

- "host": <string>: AWS AppSync GraphQL エンドポイントまたはカスタムドメイン名のホスト。

例

```
{

 "Authorization":"M0UzQzM1MkQtMkI0Ni00OTZCLUI1NkQtMUM0MTQ0QjVBRTczCkI1REEzRTIxLTk5NzItNDJENi1BQ
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

クエリ文字列によるヘッダー

まず、 hostと を含む JSON オブジェクトAuthorizationが文字列に変換されます。次に、この文字列は base64 エンコードを使用してエンコードされます。結果の base64 エンコードされた文字列は、 AWS AppSync リアルタイムエンドポイントとの接続を確立するために header WebSocket URL に という名前のクエリパラメータとして追加されます。生成されるリクエスト URL の形式は次のとおりです。

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

base64 でエンコードされたヘッダーオブジェクトに加えて、空の JSON オブジェクト {} も base64 でエンコードされ、WebSocket URL payloadに という名前の別のクエリパラメータとして含まれることに注意してください。

経由のヘッダー **Sec-WebSocket-Protocol**

host と を含む JSON オブジェクトAuthorizationは文字列に変換され、base64Url エンコードを使用してエンコードされます。結果の base64Url-encodedされた文字列には、 というプレフィックスが付けられますheader-。このプレフィックス付き文字列は、 AWS AppSync リアルタイムエンドポイントとの WebSocket 接続を確立するときに、 graphql-wsSec-WebSocket-Protocolヘッダーの に加えて新しいサブプロトコルとして使用されます。

生成されるリクエスト URL の形式は次のとおりです。

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

Sec-WebSocket-Protocol ヘッダーには次の値が含まれます。

```
"sec-websocket-protocol" : ["graphql-ws", "header-
ewogICAgImhvc3QiOiJleGFtcGxlMTIzNDU2Nzg5MDAwMC5hcHBzeW5jLWFwaS51cy1lYXN0LTEuYW1hem9uYXdzLmNvbSI
```

標準 HTTP ヘッダー経由のヘッダー

この方法では、 AWS AppSync リアルタイムエンドポイントとの WebSocket 接続を確立するときに、ホストと認可の情報が標準の HTTP ヘッダーを使用して送信されます。生成されるリクエスト URL の形式は次のとおりです。

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

リクエストヘッダーには以下が含まれます。

```
"sec-websocket-protocol" : ["graphql-ws"]
"Authorization":"eyEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJHWEFLSXBieU5WNHhsQjEXAMPLEnM2WldvPSIsImFsZy
zEE2DJH7sHOl2zxYi7f-SmEGoh2AD8emxQRYajByz-rE4Jh0QOymN2Ys-ZIkMpVBTPgu-
TMWDyOHhDUmUj2OP82yeZ3wlZAtr_gM4LzjXUXmI_K2yGjuXfXTaa1mvQEBG0mQfVd7SfwXB-
jcv4RYVi6j25qgow9Ew52ufurPqaK-3WAKG32KpV8J4-Wejq8t0c-
yA7sb8EnB551b7TU93uKRiVVK3E55Nk5ADPoam_WYE45i3s5qVAP_-InW75NUoOCGTsS8YWMfb6ecHYJ-1j-
bzA27zaT9VjctXn9byNFZmEXAMPLExw",
"host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
```

リアルタイム WebSocket オペレーション

AWS AppSync で成功した WebSocket ハンドシェイクを開始した後、クライアントは後続のメッセージを送信して、異なるオペレーションのために AWS AppSync に接続する必要があります。これらのメッセージには、次のデータが必要です。

- `type`: 操作のタイプ。
- `id`: サブスクリプションの一意の識別子。この目的のために UUID を使用することをお勧めします。
- `payload`: オペレーションタイプに応じて、関連付けられたペイロード。

`id` フィールド `type` とフィールドはオプションで、`payload` フィールドは必須です。

イベントのシーケンス

サブスクリプションリクエストを正常に開始、確立、登録、および処理するには、クライアントは次のシーケンスを実行する必要があります。

1. 接続の初期化 (connection_init)
2. 接続確認応答 (connection_ack)
3. サブスクリプション登録 (start)
4. サブスクリプション確認応答 (start_ack)
5. サブスクリプションの処理 (data)
6. サブスクリプションの登録解除 (stop)

接続初期化メッセージ

(オプション) ハンドシェイクが成功すると、クライアントはconnection_initメッセージを送信して AWS AppSync リアルタイムエンドポイントとの通信を開始できます。メッセージは、次のように JSON オブジェクトを文字列化することで取得される文字列です。

```
{ "type": "connection_init" }
```

接続確認メッセージ

connection_init メッセージを送信した後、クライアントは connection_ack メッセージを待つ必要があります。connection_ack を受信前に送信されたメッセージはすべて無視されます。メッセージは次のように表示されるはずです。

```
{
  "type": "connection_ack",
  "payload": {
    // Time in milliseconds waiting for ka message before the client should terminate
 the WebSocket connection
    "connectionTimeoutMs": 300000
  }
}
```

キープアライブメッセージ

接続確認メッセージに加えて、クライアントは定期的にキープアライブメッセージを受信します。接続タイムアウト期間内にクライアントがキープアライブメッセージを受信しない場合、クライアントは接続を閉じる必要があります。 AWS AppSync は、接続を自動的にシャットダウンするまで (24 時間後) 、これらのメッセージを送信し、登録されたサブスクリプションを処理し続けます。キープアライブメッセージはハートビートであり、クライアントによる確認は不要です。

```
{ "type": "ka" }
```

サブスクリプション登録メッセージ

クライアントがconnection_ackメッセージを受信すると、クライアントはサブスクリプション登録メッセージを AWS AppSync に送信できます。このタイプのメッセージは、以下のフィールドを含む文字列化された JSON オブジェクトです。

- サブスクリプションの ID。この ID はサブスクリプションごとに一意である必要があります。そうしないと、サーバーはサブスクリプション ID が重複していることを示すエラーを返します。
- `"type": "start"`: 定数 `<string>` パラメータ。
- `"payload": <Object>`: サブスクリプションに関連する情報が含むオブジェクト。
  - `"data": <string>`: GraphQL クエリと変数を含む文字列化された JSON オブジェクト。
    - `"query": <string>`: GraphQL オペレーション。
    - `"variables": <Object>`: クエリの変数を含むオブジェクト。
  - `"extensions": <Object>`: 認可オブジェクトを含むオブジェクト。
- `"authorization": <Object>`: 認可に必要なフィールドが含むオブジェクト。

サブスクリプション登録の認可オブジェクト

また、AWS AppSync API 認可モードに基づくヘッダーパラメータフォーマット セクションにある同じルールが認可オブジェクトに適用されます。唯一の例外は IAM で、SigV4 署名情報がわずかに異なります。詳細については、IAM の例を参照してください。

Amazon Cognito ユーザープールの使用例:

```
{
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n
 __typename\\n message\\n }\\n }\",\"variables\":{}}",
      "extensions": {
        "authorization": {
          "Authorization":
 "eyEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJEXAMPLEBieU5WNHhsQjhPVW9YMnM2WldvPSIsImFsZyI6IlEXAMPLEn0.e
qTCtrYeboUJ4luRSTPXaNewNeEXAMPLE14C6sfg05tO0fOMpiUwj9k19gtNCCMqoSsjtQoUweFnH4JYa5EXAMPLEVxOyQEQ
RWvW7yQU3sNQRLEXAMPLEcd0yufBiCYs3dfQxTTdvR1B6Wz6CD78lfNeKqfzzUn2beMoup2h6EXAMPLE4ow8cUPUPvG0DzR
          "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
```

```
        }
      }
  },
  "type": "start"
}
```

IAM の使用例:

```
{
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n
 __typename\\n message\\n }\\n }\",\"variables\":{}}",
    "extensions": {
      "authorization": {
        "accept": "application/json, text/javascript",
        "content-type": "application/json; charset=UTF-8",
        "X-Amz-Security-Token":
 "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXRoZWFEXAMPLEcwRQIgAh97Cljq7wOPL8KsxP3YtDuyc/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKEn0zyUhBEXAMPLEjj/////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUdAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtwR+9zF7NaMMMse07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZOzlZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnGlSNVx7SMcfovkFDqQamm
+88y1OwwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVYlj9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEfwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCkcCGFnwBNFLmfnbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpgqJsPKt582caFKArIFIvO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+tOLnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOKtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtKOOweDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA==",
        "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
 Signature=b90131a61a7c4318e1c35ead5dbfdeb46339a7585bbdbeceeaff51f4022eb1fd",
        "content-encoding": "amz-1.0",
        "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
        "x-amz-date": "20200401T001010Z"
```

```
      }
    }
  },
  "type": "start"
}
```

カスタムドメイン名の使用例

```
{
  "id": "key-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n
 __typename\\n message\\n }\\n }\",\"variables\":{}}",
      "extensions": {
        "authorization": {
          "x-api-key": "da2-12345678901234567890123456",
          "host": "api.example.com"
         }
      }
  },
  "type": "start"
}
```

SigV4 署名は、URL に /connect を追加する必要がなく、data は JSON 文字列化された GraphQL オペレーションに置き換えられます。次に、SigV4 署名リクエストの例を示します。

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql",
  data: "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n __typename
\\n message\\n }\\n }\",\"variables\":{}}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

サブスクリプション確認メッセージ

サブスクリプション開始メッセージを送信した後、クライアントは AWS AppSync がstart_ackメッセージを送信するのを待つ必要があります。start_ack メッセージでは、サブスクリプションが成功したことが示されます。

サブスクリプション確認の例

```
{
  "type": "start_ack",
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

エラーメッセージ

接続の初期化またはサブスクリプションの登録が失敗した場合、またはサーバーからサブスクリプションが終了した場合、サーバーはクライアントにエラーメッセージを送信します。接続の開始時間中にエラーが発生した場合、接続はサーバーによって閉じられます。

- "type": "error": 定数 <string> パラメータ。
- "id": <string>: 対応する登録済みサブスクリプションの ID (該当する場合)。
- "payload" <Object>: 対応するエラー情報を含むオブジェクト。

例:

```
{
  "type": "error",
  "payload": {
    "errors": [
      {
        "errorType": "LimitExceededError",
        "message": "Rate limit exceeded"
      }
    ]
  }
}
```

データメッセージの処理

クライアントがミューテーションを送信すると、 AWS AppSync はミューテーションに関心のあるすべてのサブスクライバーを識別し、"start"サブスクリプションオペレーションidから対応するサブスクリプションを使用して各サブスクライバーに"type":"data"メッセージを送信します。クライアントは、データメッセージを受信したときに、対応するサブスクリプションと照合できるように、送信するサブスクリプション id を追跡する必要があります。

- "type": "data": 定数 <string> パラメータ。
- "id": <string>: 対応する登録済みサブスクリプションの ID。
- "payload" <Object>: サブスクリプション情報を含むオブジェクト。

例:

```
{
  "type": "data",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": {
      "onCreateMessage": {
        "__typename": "Message",
        "message": "test"
      }
    }
  }
}
```

サブスクリプションの登録解除メッセージ

アプリがサブスクリプションイベントのリッスンを停止する場合、クライアントは次の文字列化された JSON オブジェクトを含むメッセージを送信する必要があります。

- "type": "stop": 定数 <string> パラメータ。
- "id": <string>: 登録を解除するサブスクリプションの ID。

例:

```
{
```

```
  "type":"stop",
  "id":"ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

AWS AppSync は、次の文字列化された JSON オブジェクトを含む確認メッセージを送信します。

- `"type": "complete"`: 定数 `<string>` パラメータ。
- `"id": <string>`: 登録を解除したサブスクリプションの ID。

クライアントは確認メッセージを受信すると、この特定のサブスクリプションに関するメッセージをそれ以上受信しません。

例:

```
{
  "type":"complete",
  "id":"eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

WebSocket の切断

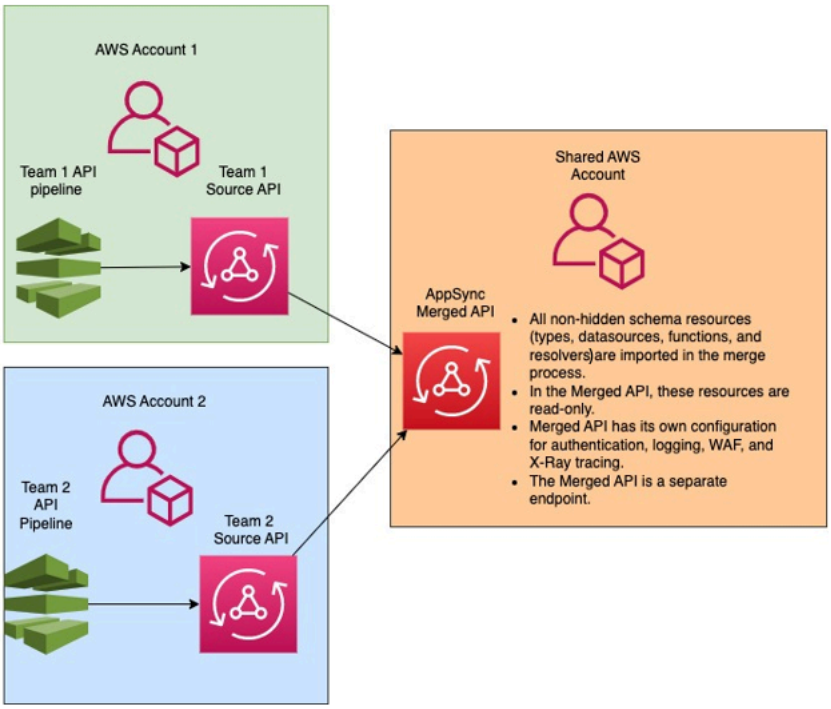
切断する前に、クライアントは、データ損失を避けるために、WebSocket 接続を介して現在オペレーションが行われていないことを確認するために必要なロジックを持っている必要があります。WebSocket から切断する前に、すべてのサブスクリプションを登録解除する必要があります。

での APIsマージ AWS AppSync

組織内で GraphQL の使用が拡大するにつれて、API の使いやすさと API の開発速度の間でトレードオフが生じる可能性があります。一方、組織は AWS AppSync と GraphQL を採用して、開発者が 1 回のネットワーク呼び出しで 1 つ以上のデータドメインのデータに安全にアクセスし、操作し、結合するために使用できる柔軟な API を提供することで、アプリケーション開発を簡素化します。一方、単一の GraphQL API エンドポイントにまとめられたさまざまなデータドメインを担当する組織内のチームは、開発速度を上げるために、API アップデートを互いに独立して作成、管理、およびデプロイする機能を求める場合があります。

この緊張を解決するために、 AWS AppSync Merged APIs機能を使用すると、さまざまなデータドメインのチームが個別に AWS AppSync APIs (GraphQL スキーマ、リゾルバー、データソース、関数など) を作成してデプロイし、1 つのマージされた API に結合できます。これにより、組織は使いや

すいクロスドメイン API を維持できるようになり、その API に貢献するさまざまなチームが API を迅速かつ独立して更新できるようになります。



Merged APIs を使用すると、組織は複数の独立したソース AWS AppSync APIsのリソースを単一の AWS AppSync Merged API エンドポイントにインポートできます。これを行うには、 AWS AppSync を使用して AWS AppSync ソース APIs のリストを作成し、スキーマ、タイプ、データソース、リゾルバー、関数など、ソース APIs に関連付けられたすべてのメタデータを新しいマージされた API に AWS AppSync マージできます。

マージ中は、複数のスキーマを組み合わせる際の型名の競合など、ソース API のデータ内容の不一致が原因でマージによる競合が発生する可能性があります。ソース API の定義が競合しない単純なユースケースでは、ソース API スキーマを変更する必要はありません。結果の Merged API は、元の AWS AppSync APIs からすべてのタイプ、リゾルバー、データソース、関数をインポートするだけです。競合が発生する複雑なユースケースでは、ユーザー/チームはさまざまな方法で競合を解決する必要があります。 は、マージ競合を軽減できるいくつかのツールと例をユーザー AWS AppSync に提供します。

で設定された後続のマージ AWS AppSync は、ソース APIs で行われた変更を関連する Merged API に伝達します。

Merged API とフェデレーション

GraphQL コミュニティには、GraphQL スキーマを結合し、共有グラフを通じてチームコラボレーションを可能にするための多くのソリューションとパターンがあります。 AWS AppSync マージされた APIsはスキーマ構成にビルド時間のアプローチを採用しており、ソース APIs は別の Merged API に結合されます。別の方法として、ランタイムルーターを複数のソース API またはサブグラフに重ねる方法があります。このアプローチでは、ルーターはリクエストを受け取り、メタデータとして維持する結合スキーマを参照し、リクエストプランを構築し、基盤となるサブグラフ/サーバー全体にリクエスト要素を分散します。次の表は、 AWS AppSync Merged API ビルドタイムアプローチと、GraphQL スキーマ構成に対するルーターベースのランタイムアプローチを比較したものです。

Feature	AppSync Merged API	Router-based solutions
Sub-graphs managed independently	Yes	Yes
Sub-graphs addressable independently	Yes	Yes
Automated schema composition	Yes	Yes
Automated conflict detection	Yes	Yes
Conflict resolution via schema directives	Yes	Yes
Supported sub-graph servers	AWS AppSync*	Varies
Network complexity	Single, merged API means no extra network hops.	Multi-layer architecture requires query planning and delegation, sub-query parsing and serialization/deserialization, and reference resolvers in sub-graphs to perform joins.
Observability support	Built-in monitoring, logging, and tracing. A single, Merged	Build-your-own observability across router and all associated sub-graph servers. Complex

	API server means simplified debugging.	debugging across distributed system.
Authorization support	Built in support for multiple authorization modes.	Build-your-own authorization rules.
Cross account security	Built-in support for cross-AWS cloud account associations.	Build-your-own security model.
Subscriptions support	Yes	No

* AWS AppSync マージされた APIsはソース AWS AppSync APIsにのみ関連付けることができます。サブグラフ全体 AWS AppSync およびAWS AppSync サブグラフ以外のスキーマ構成のサポートが必要な場合は、1 つ以上の AWS AppSync GraphQL および/または Merged APIs をルーターベースのソリューションに接続できます。例えば、Apollo Federation v2: Apollo GraphQL Federation with AWS AppSync を使用したルーターベースのアーキテクチャを使用してサブグラフとして AWS AppSync APIs を追加するためのリファレンスブログを参照してください。

トピック

- Merged API の競合の解決
- スキーマの設定
- 承認モードの設定
- 実行ロールの設定
- AWS RAMを使用したクロスアカウント Merged API の設定
- マージ
- Merged API に対するその他のサポート
- Merged API の制限
- Merged API の作成

Merged API の競合の解決

マージの競合が発生した場合、 は問題のトラブルシューティングに役立ついくつかのツールと例をユーザー AWS AppSync に提供します (複数可) 。

Merged API スキーマのディレクティブ

AWS AppSync は、ソース API 間の競合を軽減または解決するために使用できるいくつかの GraphQL ディレクティブを導入しました。 APIs

- @canonical: このディレクティブは、名前とデータが似ている型/フィールドの優先順位を設定します。2 つ以上のソース API が同じ GraphQL タイプまたはフィールドを持つ場合、いずれかの API がタイプまたはフィールドに標準として注釈を付けることができ、マージ時に優先順位が付けられます。他のソース API でこのディレクティブでアノテーションが付けられていない競合するタイプ/フィールドは、マージ時に無視されます。
- @hidden: このディレクティブは特定の型/フィールドをカプセル化して、マージプロセスから削除します。チームは、内部クライアントだけが特定の型付きデータにアクセスできるように、ソース API の特定のタイプや操作を削除または非表示にしたい場合があります。このディレクティブをアタッチすると、タイプやフィールドは Merged API にマージされません。
- @renamed: このディレクティブは、名前の競合を減らすためにタイプ/フィールドの名前を変更します。異なる API が同じタイプまたはフィールド名を持つ場合があります。ただし、これらはすべて、マージされたスキーマで使用可能である必要があります。これらすべてを Merged API に含める簡単な方法は、フィールドの名前を似ているが別の名前に変更することです。

ユーティリティスキーマのディレクティブを示すには、次の例を考えます。

この例では、2 つのソース API をマージすると仮定します。投稿 (コメントセクションやソーシャルメディア投稿など) を作成および取得する 2 つのスキーマが与えられます。タイプとフィールドが似ていると、マージ操作中に競合が発生する可能性が高くなります。以下のスニペットは、各スキーマのタイプとフィールドを示しています。

Source1.graphQL という名前の最初のファイルは、ユーザーが `putPost` ミューテーションを使用して `Posts` を作成できるようにする GraphQL スキーマです。各 `Post` にはタイトルと ID が含まれています。ID は `User`、または投稿者の情報 (メールと住所)、`Message`、またはペイロード (コンテンツ) を参照するために使用されます。`User` タイプには @canonical タグのアノテーションが付けられています。

```
# This snippet represents a file called Source1.graphql

type Mutation {
    putPost(id: ID!, title: String!): Post
}

type Post {
```

```
    id: ID!
    title: String!
}

type Message {
   id: ID!
   content: String
}

type User @canonical {
   id: ID!
   email: String!
   address: String!
}

type Query {
    singlePost(id: ID!): Post
    getMessage(id: ID!): Message
}
```

2 つ目のファイルは Source2.graphQL と呼ばれ、Source1.graphql とよく似た処理を行う GraphQL スキーマです。ただし、各タイプのフィールドは異なることに注意してください。これら 2 つのスキーマをマージすると、これらの違いによりマージによる競合が発生します。

また、Source2.graphQL には、これらの競合を減らすためのディレクティブがいくつか含まれていることにも注意してください。Post タイプには @hidden タグのアノテーションが付いており、マージ操作中にわかりにくくなります。Message タイプには @renamed タグのアノテーションが付いており、別の Message タイプと名前が競合した場合にタイプ名が ChatMessage に変更されます。

```
# This snippet represents a file called Source2.graphql

type Post @hidden  {
    id: ID!
    title: String!
    internalSecret: String!
}

type Message @renamed(to: "ChatMessage") {
   id: ID!
   chatId: ID!
   from: User!
```

```
   to: User!
}

# Stub user so that we can link the canonical definition from Source1
type User {
   id: ID!
}

type Query {
    getPost(id: ID!): Post
    getMessage(id: ID!): Message @renamed(to: "getChatMessage")
}
```

マージが行われると、結果は MergedSchema.graphql ファイルを生成します。

```
# This snippet represents a file called MergedSchema.graphql

type Mutation {
    putPost(id: ID!, title: String!): Post
}

# Post from Source2 was hidden so only uses the Source1 definition.
type Post {
    id: ID!
    title: String!
}

# Renamed from Message to resolve the conflict
type ChatMessage {
   id: ID!
   chatId: ID!
   from: User!
   to: User!
}

type Message {
   id: ID!
   content: String
}

# Canonical definition from Source1
type User {
   id: ID!
```

```
    email: String!
    address: String!
}

type Query {
    singlePost(id: ID!): Post
    getMessage(id: ID!): Message

    # Renamed from getMessage
    getChatMessage(id: ID!): ChatMessage
}
```

マージでは次のことが起こりました。

- @canonical アノテーションにより、Source1.graphql からの User タイプが Source2.graphql からの User よりも優先されました。
- Source1.graphql からの Message タイプがマージに含まれました。ただし、Source2.graphql からの Message には命名上の競合がありました。その @renamed アノテーションにより、代替名 ChatMessage でマージに含まれました。
- Source1.graphql からPost タイプは含まれていましたが、Source2.graphql からの Post タイプは含まれていませんでした。通常、このタイプでは競合が発生しますが、Source2.graphql からの Post タイプには @hidden アノテーションが付いていたため、そのデータは難読化され、マージには含まれませんでした。この結果、競合は発生しませんでした。
- Query タイプは両方のファイルの内容を含むように更新されました。ただし、このディレクティブにより、1 つの GetMessage クエリの名前が GetChatMessage に変更されました。これにより、同じ名前の 2 つのクエリ間の名前の競合が解決されました。

また、競合するタイプにディレクティブが追加されない場合もあります。この場合、マージされたタイプには、そのタイプのすべてのソース定義のすべてのフィールドの和集合が含まれます。例えば、次の例を考えてみます。

このスキーマは Source1.graphql と呼ばれ、Posts の作成と取得を可能にします。設定は、前述のものと似ていますが、情報が少なくなっています。

```
# This snippet represents a file called Source1.graphql

type Mutation {
    putPost(id: ID!, title: String!): Post
}
```

```
type Post  {
    id: ID!
    title: String!
}

type Query {
    getPost(id: ID!): Post
}
```

このスキーマは Source2.graphql と呼ばれ、Reviews (映画の評価やレストランのレビューなど) の作成と取得が可能です。Reviews は、同じ ID 値の Post と関連付けられています。これらには、レビュー投稿のタイトル、投稿 ID、ペイロードメッセージが含まれます。

マージすると、この 2 つの Post タイプが競合することになります。この問題を解決するアノテーションがないため、デフォルトの動作では、競合するタイプに対してユニオンオペレーションを実行します。

```
# This snippet represents a file called Source2.graphql

type Mutation {
    putReview(id: ID!, postId: ID!, comment: String!): Review
}

type Post  {
    id: ID!
    reviews: [Review]
}

type Review {
   id: ID!
   postId: ID!
   comment: String!
}

type Query {
    getReview(id: ID!): Review
}
```

マージが行われると、その結果は MergedSchema.graphql ファイルを生成します。

```
# This snippet represents a file called MergedSchema.graphql
```

```
type Mutation {
    putReview(id: ID!, postId: ID!, comment: String!): Review
    putPost(id: ID!, title: String!): Post
}

type Post  {
    id: ID!
    title: String!
    reviews: [Review]
}

type Review {
   id: ID!
   postId: ID!
   comment: String!
}

type Query {
    getPost(id: ID!): Post
    getReview(id: ID!): Review
}
```

マージでは次のようなことが起こりました。

- Mutation タイプは競合せず、マージされました。
- Post タイプフィールドはユニオンオペレーションによって結合されました。この 2 つのユニオンによって、単一の id、title、および単一の reviews が生成されることに注目してください。
- Review のタイプは競合せず、マージされました。
- Query のタイプは競合せず、マージされました。

共有タイプのリゾルバーの管理

上記の例で、Source1.graphql が PostDatasource という名前の DynamoDB データソースを使用する、Query.getPost 上のユニットリゾルバーを構成している場合を考えてみましょう。このリゾルバーは Post タイプの id と title を返します。ここで、Source2.graphql が Post.reviews でパイプラインリゾルバーを設定し、2 つの関数を実行しているとします。Function1 には、カスタムの権限チェックを実行するための None データソースがアタッチされています。Function2 には reviews テーブルをクエリするための DynamoDB データソースがアタッチされています。

```
query GetPostQuery {
    getPost(id: "1") {
        id,
        title,
        reviews
    }
}
```

上記のクエリがクライアントによって Merged API エンドポイントに対して実行されると、 AWS AppSync サービスはまず Query.getPostから のユニットリゾルバーを実行しSource1、 を呼び出しPostDatasourceて DynamoDB からデータを返します。次に、Post.reviews パイプラインリゾルバーを実行し、そこで Function1 がカスタム認可ロジックを実行して、Function2 が $context.source で見つかった id に付与されたレビューを返します。サービスはリクエストを単一の GraphQL 実行として処理し、この単純なリクエストには 1 つのリクエストトークンのみが必要です。

共有タイプのリゾルバー競合の管理

Source2 のフィールドリゾルバーを超えて一度に複数のフィールドを提供するために、Query.getPost にリゾルバーも実装する場合を考えてみよう。Source1.graphql は以下のようになるかもしれません。

```
# This snippet represents a file called Source1.graphql

type Post  {
    id: ID!
    title: String!
    date: AWSDateTime!
}

type Query {
    getPost(id: ID!): Post
}
```

Source2.graphql は以下のようになるかもしれません。

```
# This snippet represents a file called Source2.graphql

type Post  {
  id: ID!
```

```
  content: String!
  contentHash: String!
  author: String!
}

type Query {
    getPost(id: ID!): Post
}
```

Merged APIs では複数のソースリゾルバーを同じフィールドにアタッチできないため、これら 2 つのスキーマをマージしようとすると AWS AppSync マージエラーが発生します。この競合を解決するには、Source2.graphql に、Post タイプから所有するフィールドを定義する別の型を追加するように要求するフィールドリゾルバーパターンを実装できます。次の例では、PostInfo というタイプを追加します。このタイプには、Source2.graphql によって解決されるコンテンツフィールドと作成者フィールドが含まれます。Source1.graphql は Query.getPost にアタッチされたリゾルバーを実装し、Source2.graphQL はすべてのデータを正常に取得できるようにリゾルバーを Post.postInfo にアタッチします。

```
type Post  {
  id: ID!
  postInfo: PostInfo
}

type PostInfo {
   content: String!
   contentHash: String!
   author: String!
}

type Query {
    getPost(id: ID!): Post
}
```

このような競合を解決するには、ソース API スキーマを書き直す必要があり、場合によってはクライアントがクエリを変更する必要がありますが、このアプローチの利点は、マージされたリゾルバーの所有権がソースチーム間で明確になることです。

スキーマの設定

Merged API を作成するためのスキーマの設定は、次の 2 つの関係者が担当します。

- Merged API 所有者 - Merged API 所有者は、Merged API の承認ロジックと、ロギング、トレーシング、キャッシュ、WAF サポートなどの詳細設定を構成する必要があります。
- 関連ソース API 所有者 - 関連 API 所有者は、Merged API を構成するスキーマ、リゾルバー、データソースを設定する必要があります。

Merged API のスキーマは関連するソース API のスキーマから作成されるため、読み取り専用です。つまり、スキーマの変更はソース API で開始する必要があります。 AWS AppSync コンソールでは、スキーマウィンドウの上にあるドロップダウンリストを使用して、Merged APIs スキーマと Merged API に含まれるソース API の個々のスキーマを切り替えることができます。

承認モードの設定

Merged API を保護するために複数の承認モードを使用できます。の認可モードの詳細については AWS AppSync、「認可と認証」を参照してください。

Merged API では以下の承認モードを使用できます。

- API キー: 最も単純な商人戦略。すべてのリクエストには、`x-api-key` リクエストヘッダーの下に API キーを含める必要があります。期限切れの API キーは、有効期限後 60 日間保管されます。
- AWS Identity and Access Management (IAM): AWS IAM 認可戦略は、sigv4 で署名されたすべてのリクエストを承認します。
- Amazon Cognito ユーザープール: Amazon Cognito ユーザープールを使用してユーザーを承認すると、よりきめ細かい制御が可能になります。
- AWS Lambda Authorizer: カスタムロジックを使用して AWS AppSync API へのアクセスを認証および認可できるサーバーレス関数。
- OpenID Connect: この認可タイプは、OIDC 準拠サービスによって提供される OpenID Connect (OIDC) トークンを適用します。アプリケーションは、アクセス制御に対して、使用する OIDC プロバイダーによって定義されたユーザーと権限を活用できます。

Merged API の承認モードは、統合された API の所有者によって設定されます。マージ操作時に、Merged API には、ソース API に設定されたプライマリ認可モードを、独自のプライマリ認可モードまたはセカンダリ認可モードとして含める必要があります。そうしないと、互換性がなくなり、マージオペレーションは競合が発生して失敗します。ソース API でマルチ認証ディレクティブを使用すると、マージプロセスでこれらのディレクティブを統合エンドポイントに自動的にマージできます。ソース API のプライマリ承認モードがMerged API のプライマリ承認モードと一致しない場

合、ソース API のタイプの承認モードの一貫性が保たれるように、これらの承認ディレクティブが自動的に追加されます。

実行ロールの設定

Merged API を作成するときは、サービスロールを定義する必要があります。 AWS サービスロールは、ユーザーに代わってタスクを実行するために AWS サービスで使用される AWS Identity and Access Management (IAM) ロールです。

この場合、Merged API は、ソース API に設定されたデータソースのデータにアクセスするリゾルバーを実行する必要があります。そのために必要なサービスロールは `mergedApiExecutionRole` であり、`appsync:SourceGraphQL` IAM アクセス許可を介してマージされた API に含まれるソース API のリクエストを実行するための明示的なアクセス権が必要です。GraphQL リクエストの実行中に、 AWS AppSync サービスはこのサービスロールを引き受け、`appsync:SourceGraphQL`アクションを実行するロールを承認します。

AWS AppSync では、IAM APIs。最non-top-levelフィールドの場合、 ではソース API ARN 自体に対するアクセス許可を定義 AWS AppSync する必要があります。Merged API の特定の非トップレベルフィールドへのアクセスを制限するには、Lambda 内にカスタムロジックを実装するか、@hidden ディレクティブを使用してソース API フィールドを Merged API から非表示にすることをお勧めします。ソース API 内のすべてのデータ操作をロールに実行させたい場合は、以下のポリシーを追加できます。最初のリソースエントリはすべての最上位フィールドへのアクセスを許可し、2 番目のエントリはソース API リソース自体を認可する子リゾルバーを対象としていることに注意してください。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [ "appsync:SourceGraphQL"],
        "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId/
*",
            "arn:aws:appsync:us-west-2:123456789012:apis/
YourSourceGraphQLApiId"]
    }]
}
```

アクセスを特定の最上位フィールドのみに制限したい場合は、次のようなポリシーを使用できます。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [ "appsync:SourceGraphQL"],
        "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId/
types/Query/fields/<Field-1>",
            "arn:aws:appsync:us-west-2:123456789012:apis/
YourSourceGraphQLApiId"]
    }]
}
```

AWS AppSync コンソール API 作成ウィザードを使用してサービスロールを生成し、マージされた API がマージされた API と同じアカウントにあるソース APIs で設定されたリソースにアクセスすることを許可することもできます。ソース APIs がマージされた API と同じアカウントにない場合は、まず Resource Access Manager () を使用して AWS リソースを共有する必要がありますAWS RAM。

AWS RAMを使用したクロスアカウント Merged API の設定

Merged API を作成するときは、オプションで、 AWS Resource Access Manager () を介して共有されている他のアカウントのソース APIs を関連付けることができますAWS RAM。 AWS RAM これにより、 AWS アカウント間、組織内または組織単位 (OUs)、IAM ロールとユーザーとリソースを安全に共有できます。

AWS AppSync は と統合 AWS RAM され、単一の Merged APIs から複数のアカウントにわたるソース API の設定とアクセスをサポートします。 AWS RAM では、リソース共有、またはリソースのコンテナ、およびリソースごとに共有されるアクセス許可セットを作成できます。リソース共有に AWS AppSync APIs を追加できます AWS RAM。リソース共有内で、 AWS AppSync は RAM の AWS AppSync API に関連付けることができる 3 つの異なるアクセス許可セットを提供します。

1. `AWSRAMPermissionAppSyncSourceApiOperationAccess`: 他のアクセス許可が指定され AWS RAM ていない場合に で AWS AppSync API を共有するときに追加されるデフォルトのアクセス許可セット。このアクセス許可セットは、ソース AWS AppSync API

を Merged API 所有者と共有するために使用します。このアクセス許可には、ソース API `appsync:AssociateMergedGraphqlApi` に対する権限と、ランタイムにソース API リソースにアクセスするために必要な `appsync:SourceGraphQL` アクセス許可が含まれます。

2. `AWSRAMPermissionAppSyncMergedApiOperationAccess`: このアクセス許可セットは、Merged API をソース API 所有者と共有する場合に設定する必要があります。このアクセス許可セットにより、ソース API は Merged API を設定できるようになります。これには、ターゲットプリンシパルが所有するソース API を Merged API に関連付ける機能や、Merged API のソース API 関連付けを読み取って更新する機能が含まれます。
3. `AWSRAMPermissionAppSyncAllowSourceGraphQLAccess`: このアクセス許可セットでは、 AWS AppSync API でアクセス`appsync:SourceGraphQL`許可を使用できます。これは、ソース API を Merged API の所有者と共有するためのものです。ソース API 操作アクセス用のデフォルトの権限セットとは対照的に、この権限セットにはランライムアクセス許可 `appsync:SourceGraphQL` のみが含まれます。Merged API オペレーションへのアクセス権をソース API 所有者と共有することを選択した場合、Merged API エンドポイントを通じてランタイムアクセスできるようにするには、ソース API からのこの権限を Merged API 所有者にも共有する必要があります。

AWS AppSync は、カスタマー管理のアクセス許可もサポートしています。提供された AWS管理アクセス許可の 1 つが機能しない場合は、独自のカスタマー管理アクセス許可を作成できます。カスタマー管理アクセス許可は、 を使用して共有されたリソースで実行できる条件を正確に指定することで、作成および維持する管理アクセス許可です AWS RAM。 AWS AppSync では、独自のアクセス許可を作成するときに、次のアクションから選択できます。

1. `appsync:AssociateSourceGraphqlApi`
2. `appsync:AssociateMergedGraphqlApi`
3. `appsync:GetSourceApiAssociation`
4. `appsync:UpdateSourceApiAssociation`
5. `appsync:StartSchemaMerge`
6. `appsync:ListTypesByAssociation`
7. `appsync:SourceGraphQL`

でソース API または Merged API を適切に共有 AWS RAM し、必要に応じてリソース共有の招待が承諾されると、Merged API でソース API の関連付けを作成または更新するときに AWS AppSync コンソールに表示されます。によって提供される `ListGraphqlApis`オペレーションを

呼び出し、OTHER_ACCOUNTS所有者フィルターを使用して、アクセス許可セットに関係なく AWS AppSync 、 を使用してアカウント AWS RAM と共有されたすべての AWS AppSync APIs を一覧表示することもできます。

> **Note**
>
> 経由で共有する AWS RAM には、 の呼び出し元 AWS RAM に、共有されている API に対してappsync:PutResourcePolicyアクションを実行するアクセス許可が必要です。

マージ

マージの管理

Merged APIsは、統合 AWS AppSync エンドポイントでのチームコラボレーションをサポートすることを目的としています。チームがバックエンドで独自の独立したソース GraphQL API を独自に進化させながら、 AWS AppSync サービスが単一の Merged API エンドポイントへのリソースの統合を管理することで、コラボレーションにおける摩擦を減らし、開発リードタイムを短縮できます。

自動マージ

AWS AppSync Merged APIs に関連付けられたソース API は、ソース API に変更を加えた後に Merged API に自動的にマージ (自動マージ) するように設定できます。これにより、ソース API からの変更が常にバックグラウンドで Merged API エンドポイントに反映されます。ソース API スキーマの変更は、Merged API の既存の定義とのマージによる競合が発生しない限り、Merged API でも更新されます。ソース API の更新によってリゾルバー、データソース、または関数を更新する場合、インポートされたリソースも更新されます。自動的に解決 (自動解決) できない新しい競合が発生すると、マージ操作中にサポートされていないコンフリクトが発生したため、Merged API スキーマの更新は拒否されます。エラーメッセージは、ステータスが MERGE_FAILED の各ソース API アソシエーションのコンソールに表示されます。 AWS SDK または CLI を使用して、特定のソース API 関連付けの AWS GetSourceApiAssociationオペレーションを呼び出すことで、エラーメッセージを調べることもできます。

```
aws appsync get-source-api-association --merged-api-identifier <Merged API ARN> --
association-id <SourceApiAssociation id>
```

これにより、次の形式で結果が生成されます。

```
{
```

```
    "sourceApiAssociation": {
        "associationId": "<association id>",
        "associationArn": "<association arn>",
        "sourceApiId": "<source api id>",
        "sourceApiArn": "<source api arn>",
        "mergedApiArn": "<merged api arn>",
        "mergedApiId": "<merged api id>",
        "sourceApiAssociationConfig": {
            "mergeType": "MANUAL_MERGE"
        },
        "sourceApiAssociationStatus": "MERGE_FAILED",
        "sourceApiAssociationStatusDetail": "Unable to resolve conflict on object with
 name title: Merging is not supported for fields with different types."
    }
}
```

手動マージ

ソース API のデフォルト設定は手動マージです。Merged APIs が最後に更新されてからソース API で発生した変更をマージするには、ソース API 所有者は AWS AppSync コンソールから、または AWS SDK と AWS CLI で利用可能な`StartSchemaMerge`オペレーションを介して手動マージを呼び出すことができます。

Merged API に対するその他のサポート

サブスクリプションの設定

GraphQL スキーマ構成に対するルーターベースのアプローチとは異なり、 AWS AppSync マージされた APIsは GraphQL サブスクリプションの組み込みサポートを提供します。関連するソース API で定義されたすべてのサブスクリプション操作は、変更なしでMerged API に自動的にマージされて機能します。がサーバーレス WebSockets 接続を介してサブスクリプション AWS AppSync をサポートする方法の詳細については、「リアルタイムデータ」を参照してください。 WebSockets

オブザーバビリティの設定

AWS AppSync Merged APIsは、Amazon CloudWatch を介して組み込みのログ記録、モニタリング、メトリクスを提供します。 AWS AppSync また、 は、 を介してトレースするための組み込みサポートも提供しますAWS X-Ray。

カスタムドメインの設定

AWS AppSync Merged APIsは、Merged API の GraphQL およびリアルタイムエンドポイントでカスタムドメインを使用するための組み込みサポートを提供します。

キャッシュの設定

AWS AppSync Merged APIsは、オプションでリクエストレベルやリゾルバーレベルのレスポンスをキャッシュしたり、レスポンスを圧縮したりするための組み込みサポートを提供します。詳細については、「キャッシュと圧縮」を参照してください。

プライベート API の設定

AWS AppSync Merged APIsは、Merged APIs の GraphQL エンドポイントとリアルタイムエンドポイントへのアクセスを、設定可能な VPC エンドポイントから発信されるトラフィックに制限する Private API の組み込みサポートを提供します。

ファイアウォールルールの設定

AWS AppSync Merged APIsは の組み込みサポートを提供するため AWS WAF、ウェブアプリケーションのファイアウォールルールを定義して APIsを保護できます。

監査ログ記録の設定

AWS AppSync Merged APIsは AWS CloudTrail の組み込みサポートを提供するため、監査ログを設定および管理できます。

Merged API の制限

マージド API を開発するときは、以下のルールに注意してください。

1. Merged API を別の Merged API のソース API にすることはできません。
2. ソース API を複数の Merged API に関連付けることはできません。
3. Merged API スキーマドキュメントのデフォルトのサイズ制限は 10 MB です。
4. Merged API に関連付けることができるソース API のデフォルト数は 10 です。ただし、マージド API に 10 個を超えるソース API が必要な場合は、制限の引き上げをリクエストできます。

Merged API の作成

コンソールで Merged API を作成するには

1. にサインイン AWS Management Console し、 AWS AppSync コンソールを開きます。
   - ダッシュボードで、[API の作成] を選択します。
2. [Merged API] を選択し、[次へ] を選択します。
3. [API の詳細を指定] ページで、次の情報を入力します。
   a. [認証情報] で以下の情報を入力します。
      i. マージド API の API 名を指定します。このフィールドは、他の GraphQL API と簡単に区別できるように、GraphQL API にラベルを付ける方法です。
      ii. 連絡先の詳細を指定してください。このフィールドはオプションで、GraphQL API に名前またはグループをアタッチします。他のリソースにリンクされたり生成されたりすることはなく、API 名フィールドとほとんど同じように機能します。
   b. サービスロールでは、 が実行時にリソース AWS AppSync を安全にインポートして使用できるように、マージされた API に IAM 実行ロールをアタッチする必要があります。新しいサービスロールを作成して使用することもできます。これにより、 AWS AppSync が使用するポリシーとリソースを指定できます。[既存のサービスロールを使用する] を選択し、ドロップダウンリストからロールを選択することで、既存の IAM ロールをインポートすることもできます。
   c. [プライベート API 設定] では、プライベート API 機能を有効にすることを選択できます。マージド API を作成した後では、この選択を変更できないことに注意してください。プライベート API についての詳細は、「AWS AppSync プライベート API の使用」を参照してください。

   終了したら、[次へ] を選択します。
4. 次に、マージド API の基盤として使用される GraphQL API を追加する必要があります。[ソース API の選択] ページで、以下の情報を入力します。
   a. AWS アカウントテーブルAPIs で、ソース APIs の追加を選択します。GraphQL API のリストでは、各エントリには次のデータが含まれます。
      i. 名前: GraphQL API の API 名フィールド。
      ii. API ID: GraphQL API の一意の ID 値。
      iii. プライマリ承認モード: GraphQL API のデフォルト承認モード。 AWS AppSyncでの認可モードについての詳細は、「認可と認証」を参照してください。
      iv. 追加認可モード: GraphQL API で設定されていたセカンダリ認可モード。

v. API の名前フィールドの横にあるチェックボックスを選択して、マージド API で使用する API を選択します。次に、[ソース API を追加] を選択します。選択した GraphQL API は、 AWS アカウントテーブルの API テーブルに表示されます。

b. 他の AWS アカウントの APIsで、ソース APIs の追加を選択します。このリストの GraphQL APIs は、 AWS Resource Access Manager () を通じてリソースを共有している他のアカウントから取得されますAWS RAM。この表の GraphQL API を選択するプロセスは、前のセクションのプロセスと同じです。を通じてリソースを共有する方法の詳細については AWS RAM、「 とは AWS Resource Access Manager」を参照してください。

終了したら、[次へ] を選択します。

c. プライマリ承認モードを追加します。詳細については、「認可と認証」を参照してください。[次へ] をクリックします。

d. 入力内容を確認し、[API を作成] を選択します。

RDS イントロスペクションを使用した GraphQL API の構築

AWS AppSyncのイントロスペクションユーティリティは、データベーステーブルからモデルを検出し、GraphQL の型を提案できます。 AWS AppSync コンソールの API 作成ウィザードは、Aurora MySQL または PostgreSQL データベースから API を即座に生成できます。データを読み書きするための型と JavaScript リゾルバーを自動的に作成されます。

AWS AppSync は、Amazon RDS Data API を介して Amazon Aurora データベースと直接統合します。Amazon RDS Data API は、永続的なデータベース接続を必要とするのではなく、SQLステートメントを実行するために AWS AppSync に接続する安全な HTTP エンドポイントを提供します。これを使用して、Aurora で MySQL および PostgreSQL ワークロードのリレーショナルデータベース API を作成できます。

でリレーショナルデータベース用の API を構築する AWS AppSync と、いくつかの利点があります。

- データベースはクライアントに直接公開されないため、アクセスポイントをデータベース自体から切り離すことができます。
- さまざまなアプリケーションのニーズに合わせた専用の API を構築できるため、フロントエンドでのカスタムビジネスロジックが不要になります。これは Backend For Frontend (BFF) パターンと一致します。

- 認可とアクセスコントロールは、アクセスを制御するためにさまざまな認可モードを使用して AWS AppSync レイヤーに実装できます。データベースに接続するために、ウェブサーバーのホスティングやプロキシ接続など、追加のコンピューティングリソースは必要ありません。
- リアルタイム機能はサブスクリプションを通じて追加でき、AppSync を介して行われたデータミューテーションは、接続されたクライアントに自動的にプッシュされます。
- クライアントは 443 などの共通ポートを使用して HTTPS 経由で API に接続できます。

AWS AppSync により、既存のリレーショナルデータベースから APIs を簡単に構築できます。そのイントロスペクションユーティリティは、データベーステーブルからモデルを検出し、GraphQL の型を提案できます。 AWS AppSync コンソールの API 作成ウィザードを使用すると、Aurora MySQL または PostgreSQL データベースから API を即座に生成できます。データを読み書きするための型と JavaScript リゾルバーを自動的に作成されます。

AWS AppSync は、リゾルバーでの SQL ステートメントの記述を簡素化する統合 JavaScript ユーティリティを提供します。動的値を持つ静的ステートメントには AWS AppSyncの`sql`タグテンプレートを使用するか、プログラムでステートメントを構築するための`rds`モジュールユーティリティを使用できます。詳細については、「RDS のリゾルバー関数リファレンス」データソースと「組み込みモジュール」をご覧ください。

イントロスペクション機能を使用する (コンソール)

詳細なチュートリアルと入門ガイドについては、「チュートリアル: Aurora PostgreSQL Serverless で Data API を使用する」を参照してください。

AWS AppSync コンソールを使用すると、Data API で設定された既存の Aurora データベースから AWS AppSync GraphQL API を数分で作成できます。これにより、データベース設定に基づいて運用スキーマが迅速に生成されます。API をそのまま使用することも、API を基にして機能を追加することもできます。

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   - ダッシュボードで、[API の作成] を選択します。
2. [API のオプション] で、[GraphQL API]、[Amazon Aurora クラスターから始める]、[次へ] の順に選択します。
   a. [API 名] を入力します。これはコンソールで API の識別子として使用されます。
   b. 連絡先情報については、API のマネージャーを特定する連絡先を入力できます。これはオプションのフィールドです。

c. [Private API 設定] で、プライベート API 機能を有効にできます。プライベート API には、設定された VPC エンドポイント (VPCE) からのみアクセスできます。詳細については、「プライベート DNS」を参照してください。

この例では、この機能を有効にすることはお勧めしません。[次へ] 選択して入力を確認します。

3. [データベース] ページで [データベースを選択] を選択します。

a. クラスターからデータベースを選択する必要があります。最初のステップは、クラスターが存在する [リージョン] を選択することです。

b. ドロップダウンリストから [Aurora クラスター] を選択します。リソースを使用する前に、対応するData API を作成して有効にしておく必要があることに注意してください。

c. 次に、データベースの認証情報をサービスに追加する必要があります。これは主に を使用して行われます AWS Secrets Manager。シークレットが存在する [リージョン] を選択します。シークレット情報を取得する方法の詳細については、「シークレットを検索する」または「シークレットの取得」を参照してください。

d. ドロップダウンリストからシークレットを選択します。ユーザーにはデータベースの読み取りアクセス許可が必要であることに注意してください。

4. [インポート] を選択します。

AWS AppSync は、データベースのイントロスペクション、テーブル、列、プライマリキー、インデックスの検出を開始します。検出されたテーブルが GraphQL API でサポートされるかどうかをチェックします。新しい行の作成をサポートするには、テーブルにプライマリキーが必要です。プライマリキーでは、次のように複数の columns. AWS AppSync maps テーブル列を使用してフィールドを入力できます。

データ型	フィールドタイプ
VARCHAR	String
CHAR	String
BINARY	String
VARBINARY	String
TINYBLOB	String

TINYTEXT	String
TEXT	String
BLOB	String
MEDIUMTEXT	String
MEDIUMBLOB	String
LONGTEXT	String
LONGBLOB	String
BOOL	Boolean
BOOLEAN	Boolean
BIT	Int
TINYINT	Int
SMALLINT	Int
MEDIUMINT	Int
INT	Int
INTEGER	Int
BIGINT	Int
YEAR	Int
FLOAT	Float
DOUBLE	Float
DECIMAL	Float
DEC	Float
NUMERIC	Float

DATE	AWSDate
TIMESTAMP	String
DATETIME	String
TIME	AWSTime
JSON	AWSJson
ENUM	ENUM

5. テーブルの検出が完了すると、[データベース] セクションに情報が入力されます。新しい [データベーステーブル] セクションでは、テーブルからのデータがすでに入力され、スキーマの型に変換されている場合があります。必要なデータの一部が表示されない場合は、[テーブルを追加] を選択し、表示されるモーダルでそれらの型のチェックボックスをクリックして、[追加] を選択することで確認できます。

   [データベーステーブル] セクションから型を削除するには、削除する型の横にあるチェックボックスをクリックし、[削除] を選択します。削除した型は、後で再度追加したい場合、[テーブルを追加] モーダルに表示されます。

   AWS AppSync はテーブル名をタイプ名として使用しますが、名前を変更することができます。たとえば、*##*などの複数のテーブル名をタイプ名 *Movie* に変更します。[データベーステーブル] セクションで型名を変更するには、名前を変更する型のチェックボックスをクリックし、[タイプ名] 列の鉛筆アイコンをクリックします。

   選択内容に基づいてスキーマの内容をプレビューするには、[スキーマをプレビュー] を選択します。このスキーマは空にはできないため、少なくとも 1 つのテーブルを型に変換する必要があることに注意してください。また、このスキーマのサイズは 1 MB を超えることはできません。

   - [サービスロール] で、このインポート専用の新しいサービスロールを作成するか、既存のロールを使用するかを選択します。

6. [Next (次へ)] を選択します。
7. 次に、読み取り専用 API (クエリのみ) を作成するか、データの読み取りと書き込み用の API (クエリとミューテーションを含む) を作成するかを選択します。後者は、ミューテーションによってトリガーされるリアルタイムサブスクリプションもサポートします。
8. [Next (次へ)] を選択します。

9. 選択内容を確認し、[API を作成] を選択します。 AWS AppSync は API を作成し、クエリとミューテーションにリゾルバーをアタッチします。生成された API は完全に動作し、必要に応じて拡張できます。

イントロスペクション機能の使用 (API)

StartDataSourceIntrospection イントロスペクション API を使用して、データベース内のモデルをプログラムで検出できます。コマンドの詳細については、「StartDataSourceIntrospection API の使用」を参照してください。

を使用するにはStartDataSourceIntrospection、Aurora クラスターの Amazon リソースネーム (ARN)、データベース名、シー AWS Secrets Manager クレット ARN を指定します。このコマンドはイントロスペクションプロセスを開始します。GetDataSourceIntrospection コマンドを使用して結果を取得できます。検出されたモデルのストレージ定義言語 (SDL) 文字列をコマンドが返すかどうかを指定できます。これは、検出されたモデルから直接 SDL スキーマ定義を生成する場合に便利です。

例えば、単純な Todos テーブルに次のようなデータ定義言語 (DDL) ステートメントがあるとします。

```
create table if not exists public.todos
(
id serial constraint todos_pk primary key,
description text,
due timestamp,
"createdAt" timestamp default now()
);
```

イントロスペクションは、以下のように開始します。

```
aws appsync start-data-source-introspection \
  --rds-data-api-config resourceArn=<cluster-arn>,secretArn=<secret-
arn>,databaseName=database
```

次に、GetDataSourceIntrospection コマンドを使用して結果を取得します。

```
aws appsync get-data-source-introspection \
  --introspection-id a1234567-8910-abcd-efgh-identifier \
  --include-models-sdl
```

これは次の結果を返します。


```
{
    "introspectionId": "a1234567-8910-abcd-efgh-identifier",
    "introspectionStatus": "SUCCESS",
    "introspectionStatusDetail": null,
    "introspectionResult": {
        "models": [
            {
                "name": "todos",
                "fields": [
                    {
                        "name": "description",
                        "type": {
                            "kind": "Scalar",
                            "name": "String",
                            "type": null,
                            "values": null
                        },
                        "length": 0
                    },
                    {
                        "name": "due",
                        "type": {
                            "kind": "Scalar",
                            "name": "AWSDateTime",
                            "type": null,
                            "values": null
                        },
                        "length": 0
                    },
                    {
                        "name": "id",
                        "type": {
                            "kind": "NonNull",
                            "name": null,
                            "type": {
                                "kind": "Scalar",
                                "name": "Int",
                                "type": null,
                                "values": null
                            },
                            "values": null
                        },
```

```
                        "length": 0
                    },
                    {
                        "name": "createdAt",
                        "type": {
                            "kind": "Scalar",
                            "name": "AWSDateTime",
                            "type": null,
                            "values": null
                        },
                        "length": 0
                    }
                ],
                "primaryKey": {
                    "name": "PRIMARY_KEY",
                    "fields": [
                        "id"
                    ]
                },
                "indexes": [],
                "sdl": "type todos\n{\ndescription: String\n\ndue: AWSDateTime\n\nid:
 Int!\n\ncreatedAt: AW
SDateTime\n}\n"
            }
        ],
        "nextToken": null
    }
}
```

Amplify クライアントを使用したクライアントアプリケーションの構築

任意の AWS AppSync GraphQL クライアントを使用して GraphQL API に接続できますが、Amplify v6 クライアントを強くお勧めします。Amplify は、GraphQL API 用に厳密に型指定されたクライアント SDK を自動生成するだけでなく、クライアントアプリケーションでのリアルタイムデータのサポートおよび強化された GraphQL クエリ機能も提供します。ウェブアプリケーションについては、Amplify は JavaScript クライアントを生成できます。クロスプラットフォームまたはモバイル環境をターゲットとするアプリケーションについては、Amplify は Android、iOS、および React Native に対応しています。これらのプラットフォームのクライアントコード生成について詳しくは、Amplify のドキュメントを参照してください。ここでは、JavaScript React アプリケーションを使い始めるためのガイドをご紹介します。

> **Note**
>
> 開始する前に、npm と Amazon CLI の両方をインストールして設定する必要があります。Amplify v6 クライアントを使用している場合は、このガイドに従ってください。

開始するには、以下の手順を実行します。

1. ローカルマシンで、プロジェクトのディレクトリに移動します。以下のコマンドを使用して、Amplify ライブラリをインストールします。

```
npm install aws-amplify
```

2. 構成ファイルをダウンロードし、プロジェクトフォルダに配置します。構成ファイルには、通常、一部の設定 (エンドポイント、リージョン、承認モードなど) が定義された config 変数が含まれます。例えば、次のようになります。

```
const config =  {
    API: {
        GraphQL: {
          endpoint: 'https://abcdefghijklmnopqrstuvwxyz.appsync-api.us-
west-2.amazonaws.com/graphql',
          region: 'us-west-2',
          defaultAuthMode: 'apiKey',
          apiKey: ''
```

```
        }
    }
};

export default config;
```

3. コードで、Amplify ライブラリと設定をインポートして Amplify を設定します。

```
import { Amplify } from 'aws-amplify';
import config from './aws-exports.js';

Amplify.configure(config);
```

   または、API 設定でスニペットを使用して Amplify を直接セットアップします。

```
import { Amplify } from 'aws-amplify';

Amplify.configure({
  API: {
    GraphQL: {
      endpoint: 'https://abcdefghijklmnopqrstuvwxyz.appsync-api.us-
west-2.amazonaws.com/graphql',
      region: 'us-west-2',
      defaultAuthMode: 'apiKey',
      apiKey: ''
    }
  }
});
```

4. Amplify ツールチェーンを使用すると、スキーマに基づいてオペレーションを自動生成できるため、手動でスクリプトを作成する手間が省けます。アプリケーションのルートディレクトリで、次の CLI コマンドを使用します。

```
npx @aws-amplify/cli codegen add --apiId <id goes here> --region <region goes here>
```

   これにより、API のスキーマがダウンロードされ、デフォルトでクライアントヘルパーコードが `src/graphql` フォルダに生成されます。API デプロイのたびに、次のコマンドを再実行して、更新された GraphQL ステートメントとタイプを生成できます。

```
npx @aws-amplify/cli codegen
```

5. Android、Swift、Flutter、JavaScript DataStore のモデルを生成できるようになりました。スキーマをダウンロードするには以下のコマンドを使用します。

```
aws appsync get-introspection-schema --api-id <id goes here> --region <region goes
 here> --format SDL schema.graphql
```

その後、アプリケーションのルートディレクトリから以下のコマンドを実行します。

```
npx @aws-amplify/cli codegen models \
--model-schema schema.graphql \
--target [android|ios|flutter|javascript|typescript] \
--output-dir ./
```

AWS AppSync の JavaScript リゾルバーチュートリアル

データソースとリゾルバーは AWS AppSync によって GraphQL リクエストを変換し、 AWS リソースから情報を取得するために使用されます。 AWS AppSync は、特定のデータソースタイプとの自動プロビジョニングと接続をサポートしています。 AWS AppSync は AWS Lambda、データソースとして Amazon DynamoDB、リレーショナルデータベース (Amazon Aurora Serverless)、Amazon OpenSearch Service、HTTP エンドポイントもサポートしています。既存の AWS リソースで GraphQL API を使用するか、データソースとリゾルバーをゼロから構築できます。以下のセクションは、より一般的な GraphQL のユースケースをチュートリアルの形式で説明することを目的としています。

トピック

- DynamoDB JavaScript リゾルバーを使用したシンプルなポストアプリケーションの作成
- での AWS Lambda リゾルバーの使用 AWS AppSync
- AWS AppSync でのローカルリゾルバーの使用
- AWS AppSyncでの GraphQL リゾルバーの組み合わせ
- AWS AppSync での Amazon OpenSearch Service リゾルバーの使用
- AWS AppSync での DynamoDB トランザクションの実行
- AWS AppSync での DynamoDB バッチオペレーションの使用
- AWS AppSync での HTTP リゾルバーの使用
- AWS AppSyncの Data API での Aurora PostgreSQL の使用

DynamoDB JavaScript リゾルバーを使用したシンプルなポストアプリケーションの作成

このチュートリアルでは、Amazon DynamoDB テーブルを にインポート AWS AppSync して接続し、独自のアプリケーションで活用できる JavaScript パイプラインリゾルバーを使用して、完全に機能する GraphQL API を構築します。

AWS AppSync コンソールを使用して、Amazon DynamoDB リソースをプロビジョニングし、リゾルバーを作成して、データソースに接続します。また、GraphQL ステートメントを使用して Amazon DynamoDB データベースへの読み取りと書き込みを行うことができ、リアルタイムデータをサブスクライブできます。

GraphQL ステートメントが Amazon DynamoDB オペレーションに変換され、レスポンスが GraphQL に変換されるように、特定のステップを完了しておく必要があります。このチュートリアルでは、いくつかの実際のシナリオおよびデータアクセスパターンを使用して、その設定手順の概要を示します。

GraphQL API の作成

で GraphQL API を作成するには AWS AppSync

1. AppSync コンソールを開き、[API を作成] を選択します。
2. [最初から設計] を選択し、[次へ] を選択します。
3. API `PostTutorialAPI` に名前を付け、[次へ] を選択します。残りのオプションはデフォルト設定値のまま、レビューページに移動し、`Create` を選択します。

AWS AppSync コンソールは新しい GraphQL API を作成します。デフォルトでは、API キー認証モードを使用します。このコンソールを使用して、残りの GraphQL API をセットアップでき、このチュートリアルの残りの部分でクエリを実行できます。

基本的な Post API の定義

ここで、GraphQL API があるので、ポストデータの基本的な作成、取得、削除を許可する基本スキーマをセットアップできます。

スキーマにデータを追加するには

1. API で [スキーマ] タブを選択します。
2. `Post` オブジェクトを追加および取得するための `Post` タイプと操作 `addPost` を定義するスキーマを作成します。[スキーマ] ペインで、内容を次のコードに置き換えます。

```
schema {
    query: Query
    mutation: Mutation
}

type Query {
    getPost(id: ID): Post
}

type Mutation {
```

```
    addPost(
        id: ID!
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}

type Post {
    id: ID!
    author: String
    title: String
    content: String
    url: String
    ups: Int!
    downs: Int!
    version: Int!
}
```

3. [Save Schema] を選択します。

Amazon DynamoDB テーブルのセットアップ

AWS AppSync コンソールは、独自の AWS リソースを Amazon DynamoDB テーブルに保存するために必要なリソースをプロビジョニングするのに役立ちます。このステップでは、投稿を保存する Amazon DynamoDB テーブルを作成します。また、後で使用するセカンダリインデックスも設定します。

Amazon DynamoDB テーブルを作成するには

1. [スキーマ] ページで、[リソースを作成] を選択します。
2. [既存のタイプを使用] を選択し、`Post` タイプを選択します。
3. [セカンダリインデックス] セクションで、[インデックスを追加] を選択します。
4. インデックス `author-index` の名前
5. `Primary key` を `author` に、`Sort` キーを `None` に設定します。
6. GraphQL の自動生成を無効にします。この例では、リゾルバーを自分で作成します。
7. [Create] (作成) を選択します。

これで、PostTable という新しいデータソースができました。サイドタブの [データソース] にアクセスすると確認できます。このデータソースを使用して、クエリとミューテーションを Amazon DynamoDB テーブルにリンクします。

addPost リゾルバー (DAmazon DynamoDB PutItem) のセットアップ

AWS AppSync が Amazon DynamoDB テーブルを認識したので、リゾルバーを定義することで、個々のクエリとミューテーションにリンクできます。最初に作成するリゾルバーは addPost リゾルバーです。このリゾルバーによって、ユーザーが Amazon DynamoDB テーブルにポストを作成できるようになります。パイプラインリゾルバーには以下のコンポーネントがあります。

- リゾルバーをアタッチする、GraphQL スキーマ内の場所。この例では、createPost 型の Mutation フィールドにリゾルバーをセットアップしています。このリゾルバーは、呼び出し元がミューテーション { addPost(...){...} } を呼び出したときに呼び出されます。
- このリゾルバーで使用するデータソース。この例では、post-table-for-tutorial DynamoDB テーブルにエントリを追加できるように、前に定義した データソースを使用します。
- リクエストハンドラー。リクエストハンドラーは、発信者からの受信リクエストを処理し、それをが DynamoDB に対して実行する手順に変換 AWS AppSync する関数です。
- レスポンスハンドラー。レスポンスハンドラーの仕事は、DynamoDB からのレスポンスに対応して、それを GraphQL で想定されているものに変換し直すことです。これは、DynamoDB でのデータのシェイプが GraphQL での Post 型と異なる場合に便利です。ただし、この例では、両方のシェイプが同じであるため、データをそのまま渡します。

リゾルバーをセットアップするには

1. API で [スキーマ] タブを選択します。
2. リゾルバー ペインで、Mutation 型の下にある addPost フィールドを探し、「アタッチ」を選択します。
3. データソースを選択し、[作成] を選択します。
4. コードエディターで、コードを次のスニペットに置き換えます。

```
import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 const item = { ...ctx.arguments, ups: 1, downs: 0, version: 1 }
 const key = { id: ctx.args.id ?? util.autoId() }
```

```
 return ddb.put({ key, item })
}

export function response(ctx) {
 return ctx.result
}
```

5. [Save] を選択します。

> **Note**
>
> このコードでは、DynamoDB リクエストを簡単に作成できる DynamoDB モジュール utils を使用します。

AWS AppSync には、 という名前の自動 ID 生成用のユーティリティが付属しています。これはutil.autoId()、新しい投稿の ID を生成するために使用されます。ID を指定しないと、ユーティリティによって自動的に ID が生成されます。

```
const key = { id: ctx.args.id ?? util.autoId() }
```

JavaScript で使用できるユーティリティの詳細については、「リゾルバーと関数用の JavaScript ランタイム機能」を参照してください。

ポストを追加する API の呼び出し

リゾルバーが設定されたので、 は受信addPostミューテーションを Amazon DynamoDB PutItemオペレーションに変換 AWS AppSync できます。ユーザーはミューテーションを実行してテーブルに何かを入れることができるようになりました。

オペレーションを実行するには

1. API で [クエリ] タブを選択します。
2. 以下のミューテーションを [クエリ] ペインに追加します。

```
mutation addPost {
  addPost(
    id: 123,
    author: "AUTHORNAME"
    title: "Our first post!"
```

```
    content: "This is our first post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. [実行] (オレンジ色の再生ボタン) を選択し、[addPost] を選択します。新しく作成されたポストの結果が、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

次の説明は、何が起こったかを示しています。

1. AWS AppSync はミューaddPostテーションリクエストを受信しました。
2. AWS AppSync はリゾルバーのリクエストハンドラーを実行します。ddb.put 関数は以下のような PutItem リクエストを作成します。

```
{
  operation: 'PutItem',
```

```
  key: { id: { S: '123' } },
  attributeValues: {
    downs: { N: 0 },
    author: { S: 'AUTHORNAME' },
    ups: { N: 1 },
    title: { S: 'Our first post!' },
    version: { N: 1 },
    content: { S: 'This is our first post.' },
    url: { S: 'https://aws.amazon.com/appsync/' }
  }
}
```

3. AWS AppSync はこの値を使用して Amazon DynamoDB `PutItem`リクエストを生成して実行します。
4. AWS AppSync は`PutItem`リクエストの結果を受け取り、GraphQL 型に変換し直しました。

```
{
    "id" : "123",
    "author": "AUTHORNAME",
    "title": "Our first post!",
    "content": "This is our first post.",
    "url": "https://aws.amazon.com/appsync/",
    "ups" : 1,
    "downs" : 0,
    "version" : 1
}
```

5. レスポンスハンドラーは結果をすぐに返します (`return ctx.result`)。
6. 最終結果は GraphQL のレスポンスに表示されます。

getPost リゾルバー (Amazon DynamoDB GetItem) のセットアップ

これで、 Amazon DynamoDB テーブルにデータを追加できるようになりました。次は、からデータを取得できるように、`getPost` クエリを設定する必要があります。そのためには、別のリゾルバーを設定します。

リソースを追加するには、次の手順に従います。

1. API で [スキーマ] タブを選択します。
2. 右側の [リゾルバー] ペインで、`Query` タイプの `getPost` フィールドを見つけて [アタッチ] を選択します。

3. データソースを選択し、[作成] を選択します。
4. コードエディタで、コードを次のスニペットに置き換えます。

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 return ddb.get({ key: { id: ctx.args.id } })
}

export const response = (ctx) => ctx.result
```

5. リゾルバーを保存します。

> **Note**
>
> このリゾルバーでは、レスポンスハンドラーに矢印関数式を使用しています。

投稿を取得する API の呼び出し

リゾルバーがセットアップされたので、 は受信`getPost`クエリを Amazon DynamoDB `GetItem`オペレーションに変換する方法 AWS AppSync を理解します。次は、先ほど作成したポストを取得するクエリを実行します。

クエリを実行するには

1. API で [クエリ] タブを選択します。
2. クエリ ペインで次のコードを追加し、投稿を作成した後にコピーした ID を使用します。

```
query getPost {
  getPost(id: "123") {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
```

```
}
```

3. [実行] (オレンジ色の再生ボタン) を選択し、[getPost] を選択します。新しく作成されたポストの結果が、クエリペインの右側にある結果ペインに表示されます。
4. Amazon DynamoDB から取得された投稿が、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "getPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

別の方法として、次の例を指定します。

```
query getPost {
  getPost(id: "123") {
    id
    author
    title
  }
}
```

getPost クエリが id、author、title のみを必要とする場合は、DynamoDB から AWS AppSyncへの不要なデータ転送を避けるため、プロジェクション式を使用して DynamoDB テーブルから必要な属性のみを指定するようにリクエスト関数を変更できます。例えば、リクエスト関数は以下のスニペットのようになっているかもしれません。

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
```

```
 return ddb.get({
  key: { id: ctx.args.id },
  projection: ['author', 'id', 'title'],
 })
}

export const response = (ctx) => ctx.result
```

SelectionSetList を getPost と一緒に使用すると、expression を表すこともできます。

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 const projection = ctx.info.selectionSetList.map((field) => field.replace('/', '.'))
 return ddb.get({ key: { id: ctx.args.id }, projection })
}

export const response = (ctx) => ctx.result
```

updatePost ミューテーション (Amazon DynamoDB UpdateItem) の作成

これで、Amazon DynamoDB 内の Post オブジェクトを作成および取得できるようになりました。次は、オブジェクトを更新できるように、新しいミューテーションを設定します。すべてのフィールドを指定する必要がある addPost ミューテーションと比較すると、このミューテーションでは変更するフィールドのみを指定できます。また、変更するバージョンを指定できる新しい expectedVersion 引数が導入されました。オブジェクトの最新バージョンを変更していることを確認する条件を設定します。そのためには、UpdateItem Amazon DynamoDB operation.sc を使用します。

リゾルバーを更新するには

1. API で [スキーマ] タブを選択します。
2. [スキーマ] ペインの Mutation タイプを次のように変更して、新しい updatePost ミューテーションを追加します。

```
type Mutation {
    updatePost(
        id: ID!,
        author: String,
        title: String,
```

```
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post

    addPost(
        id: ID
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}
```

3. [Save Schema] を選択します。
4. 右側の [リゾルバー] ペインで、Mutation 型の新しく作成された updatePost フィールドを見つけて、[アタッチ] を選択します。以下のスニペットを使用して新しいリゾルバーを作成します。

```
import { util } from '@aws-appsync/utils';
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { id, expectedVersion, ...rest } = ctx.args;
  const values = Object.entries(rest).reduce((obj, [key, value]) => {
    obj[key] = value ?? ddb.operations.remove();
    return obj;
  }, {});

  return ddb.update({
    key: { id },
    condition: { version: { eq: expectedVersion } },
    update: { ...values, version: ddb.operations.increment(1) },
  });
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type);
  }
  return result;
```

5. 加えた変更を保存します。

このリゾルバーは ddb.update を使用して Amazon DynamoDB の作成 UpdateItem を行います。Amazon DynamoDB では、項目全体が書き込まれるのではなく、特定の属性が更新されるようにします。これを行うには、Amazon DynamoDB の更新式を使用します。

ddb.update 関数はキーと更新オブジェクトを引数として取得します。次に、入力された引数の値をチェックします。値がnullに設定されたら、DynamoDB remove オペレーションを使用して、値を DynamoDB 項目から削除する必要があることを通知します。

また、新しい condition セクションがあります。条件式を使用すると、オペレーションが実行される前に、Amazon DynamoDB に既に存在するオブジェクトの状態に基づいてリクエストが成功するかどうかを AWS AppSync と Amazon DynamoDB に指示できます。この例では、Amazon DynamoDB に現在ある項目の version フィールドが expectedVersion 引数と厳密に一致する場合にのみ、UpdateItem リクエストが成功するように指示しています。項目が更新されたら、version の値をインクリメントする必要があります。これは操作機能 increment を使えば簡単に行えます。

条件式の詳細については、「条件式リファレンス」ドキュメントを参照してください。

UpdateItem リクエストの詳細については、「UpdateItem」ドキュメントと「DynamoDB モジュール」のドキュメントを参照してください。

更新式の記述方法の詳細については、「DynamoDB UpdateExpressions」のドキュメントを参照してください。

API を呼び出して投稿を更新する

新しいリゾルバーで Post オブジェクトを更新してみましょう。

オブジェクトを更新するには

1. API で [クエリ] タブを選択します。
2. 以下のミューテーションを [クエリ] ペインに貼り付けます。また、id 引数を、前にメモしておいた値に更新する必要があります。

```
mutation updatePost {
  updatePost(
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 1
```

```
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. [実行] (オレンジ色の再生ボタン) を選択し、[updatePost] を選択します。
4. Amazon DynamoDB で更新された投稿が、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "updatePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 2
    }
  }
}
```

このリクエストでは、 AWS AppSync と Amazon DynamoDB に titleフィールドと contentフィールドのみを更新するように依頼しました。その他のフィールドは元のままです (増分する version フィールドは除く)。title 属性を新しい値に設定し、ポストから content 属性を削除しています。author、urlups、downs の各フィールドは変更されません。リクエストはまったく同じままで、このミューテーションをもう一度実行してみます。次のようなレスポンスが表示されます。

```
{
  "data": {
    "updatePost": null
```

```
  },
  "errors": [
    {
      "path": [
        "updatePost"
      ],
      "data": null,
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "The conditional request failed (Service: DynamoDb, Status Code: 400,
 Request ID: 1RR3QN5F35CS8IV5VR4OQO9NNBVV4KQNSO5AEMVJF66Q9ASUAAJG)"
    }
  ]
}
```

このリクエストは、条件式が `false`と評価されるため失敗します。

1. このリクエストを最初に実行したときに、Amazon DynamoDB 内のこの投稿の `version` フィールドの値は 1 であり、`expectedVersion` 引数と一致していました。このリクエストは成功し、Amazon DynamoDB で `version` フィールドが 2 に増分されました。
2. このリクエストを 2 回目に実行したときに、Amazon DynamoDB 内のこのポストの `version` フィールドの値は 2 であり、`expectedVersion` 引数と一致していませんでした。

このパターンは通常、「楽観的ロック」と呼ばれます。

投票ミューテーションの作成 (Amazon DynamoDB UpdateItem)

`Post` タイプには、賛成票と反対票を記録できるようにする `ups` と `downs` フィールドが含まれます。ただし、現時点では API ではこれらに対して何も実行できません。投稿に賛成および反対するための、いくつかのミューテーションを追加してみましょう。

ミューテーションを追加するには

1. API で [スキーマ] タブを選択します。

2. [スキーマ] ペインの Mutation 型を変更して DIRECTION 列挙型を追加し、新しい投票ミューテーションを追加します。

```
type Mutation {
    vote(id: ID!, direction: DIRECTION!): Post
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        id: ID,
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post!
}

enum DIRECTION {
  UP
  DOWN
}
```

3. [Save Schema] を選択します。

4. 右側の [リゾルバー] ペインで、Mutation 型の新しく作成された vote フィールドを見つけて、[アタッチ] を選択します。コードを作成して次のスニペットに置き換えることで、新しいリゾルバーを作成します。

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const field = ctx.args.direction === 'UP' ? 'ups' : 'downs';
  return ddb.update({
    key: { id: ctx.args.id },
    update: {
      [field]: ddb.operations.increment(1),
      version: ddb.operations.increment(1),
    },
```

```
  });
}

export const response = (ctx) => ctx.result;
```

5. 加えた変更を保存します。

ポストに賛成および反対するための API の呼び出し

新しいリゾルバーがセットアップされたので、 は受信upvotePostまたはdownvoteミューテーションを Amazon DynamoDB UpdateItemオペレーションに変換する方法 AWS AppSync を理解します。これで、先ほど作成したポストに賛成または反対するミューテーションを実行できるようになりました。

ミューテーションを実行するには

1. API で [クエリ] タブを選択します。
2. 以下のミューテーションを [クエリ] ペインに貼り付けます。また、id 引数を、前にメモしておいた値に更新する必要があります。

```
mutation votePost {
  vote(id:123, direction: UP) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. [実行] (オレンジ色の再生ボタン) を選択し、[votePost] を選択します。
4. Amazon DynamoDB で更新された投稿が、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "vote": {
      "id": "123",
```

```
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 0,
      "version": 4
    }
  }
}
```

5. もう何回か [実行] を選択します。このクエリを実行するたびに、ups フィールドと version フィールドが 1 つずつ増加することを確認できます。
6. クエリを変更して、別の DIRECTION を呼び出してください。

```
mutation votePost {
  vote(id:123, direction: DOWN) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

7. [実行] (オレンジ色の再生ボタン) を選択し、[votePost] を選択します。

   今度は、このクエリを実行するたびに、downs フィールドと version フィールドが 1 つずつ増加することを確認できます。

deletePost リゾルバー (Amazon DynamoDB DeleteItem) のセットアップ

次に、投稿を削除するミューテーションを作成します。そのためには、DeleteItem Amazon DynamoDB オペレーションを使用します。

ミューテーションを追加するには

1. スキーマで、「スキーマ」タブを選択します。

2. [スキーマ] ペインの Mutation 型を変更して、新しい deletePost ミューテーションを追加します。

```
type Mutation {
    deletePost(id: ID!, expectedVersion: Int): Post
    vote(id: ID!, direction: DIRECTION!): Post
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        id: ID
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post!
}
```

3. 今回は、expectedVersion フィールドをオプションにしました。[スキーマを保存] を選択します。

4. 右側の [リゾルバー] ペインで、delete タイプから新しく作成された Mutation フィールドを探し、[アタッチ] を選択します。次のコードを使用して新しいリゾルバーを作成します。

```
import { util } from '@aws-appsync/utils'

import { util } from '@aws-appsync/utils';
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  let condition = null;
  if (ctx.args.expectedVersion) {
    condition = {
      or: [
        { id: { attributeExists: false } },
        { version: { eq: ctx.args.expectedVersion } },
      ],
    };
```

```
  }
  return ddb.remove({ key: { id: ctx.args.id }, condition });
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type);
  }
  return result;
}
```

> **Note**
>
> expectedVersion 引数は省略可能な引数です。呼び出し元がリクエストで expectedVersion 引数を設定していると、項目が既に削除されている場合、または Amazon DynamoDB 内の投稿の version 属性が expectedVersion と完全に一致する場合にのみ、DeleteItem リクエストが成功することを許可する条件が、テンプレートによって追加されます。この引数が省略されている場合は、DeleteItem リクエストで条件式が指定されていません。version の値や項目が Amazon DynamoDB に存在するかどうかに関係なく、成功します。
> 注意: 項目を削除する場合でも、その項目がまだ削除されていなければ、削除された項目を返すことができます。

DeleteItem リクエストの詳細については、「DeleteItem」ドキュメントを参照してください。

ポストを削除する API の呼び出し

リゾルバーがセットアップされたので、 は受信deleteミューテーションを Amazon DynamoDB DeleteItemオペレーションに変換する方法 AWS AppSync を理解します。ユーザーはミューテーションを実行してテーブル内の何かを削除できるようになりました。

ミューテーションを実行するには

1. API で [クエリ] タブを選択します。
2. 以下のミューテーションを [クエリ] ペインに貼り付けます。また、id 引数を、前にメモしておいた値に更新する必要があります。

```
mutation deletePost {
```

```
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. [実行] (オレンジ色の再生ボタン) を選択し、[deletePost] を選択します。
4. この投稿が Amazon DynamoDB から削除されます。は、Amazon DynamoDB から削除された項目の値を AWS AppSync 返すことに注意してください。この値は、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 4,
      "version": 12
    }
  }
}
```

5. この値は、この deletePost 呼び出しによって実際に Amazon DynamoDB から項目が削除された場合にのみ返されます。再度 [実行] を選択します。
6. この呼び出しは成功しますが、値は返されません。

```
{
  "data": {
    "deletePost": null
  }
}
```

7. 次は、expectedValue を指定して、ポストを削除してみましょう。まず、これまで使用してきたポストは削除したため、まず新しいポストを作成する必要があります。

8. 以下のミューテーションを [クエリ] ペインに追加します。

```
mutation addPost {
  addPost(
    id:123
    author: "AUTHORNAME"
    title: "Our second post!"
    content: "A new post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

9. [実行] (オレンジ色の再生ボタン) を選択し、[addPost] を選択します。

10新しく作成されたポストの結果が、クエリペインの右側にある結果ペインに表示されます。新しく作成されたオブジェクトの id を書き留めておきます。その値はすぐに必要になります。これは次のように表示されます。

```
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
```

```
}
```

11.それでは、expectedVersion の値が不正な投稿を削除してみましょう。以下のミューテーションを [クエリ] ペインに貼り付けます。また、id 引数を、前にメモしておいた値に更新する必要があります。

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

12.[実行] (オレンジ色の再生ボタン) を選択し、[deletePost] を選択します。以下のレスポンスが返されます。

```
{
  "data": {
    "deletePost": null
  },
  "errors": [
    {
      "path": [
        "deletePost"
      ],
      "data": null,
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
```

```
      ],
      "message": "The conditional request failed (Service: DynamoDb, Status Code:
 400, Request ID: 7083O037M1FTFRK038A4CI9H43VV4KQNSO5AEMVJF66Q9ASUAAJG)"
    }
  ]
}
```

13.このリクエストは、条件式が `false` と評価されるため失敗します。Amazon DynamoDB `version` の投稿の値が、`expectedValue`引数で指定された値と一致しません。そのオブジェクトの現在の値が、GraphQL レスポンスの `data` セクションの `errors` フィールドで返されます。`expectedVersion` を訂正して、このリクエストをもう一度試してみます。

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

14[実行] (オレンジ色の再生ボタン) を選択し、[`deletePost`] を選択します。

今回は、リクエストが成功し、Amazon DynamoDB から削除された値が返されています。

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
```

```
    }
  }
}
```

15.再度 [実行] を選択します。この呼び出しは成功しますが、そのポストが Amazon DynamoDB で既に削除されているため、今回は値が返されません。

```
{ "data": { "deletePost": null } }
```

allPost リゾルバー (Amazon DynamoDB Scan) のセットアップ

これまでのところ、API は、見たい各投稿の `id` がわかっている場合にのみ便利です。テーブル内のすべてのポストを返す新しいリゾルバーを追加してみましょう。

ミューテーションを追加するには

1. API で [スキーマ] タブを選択します。
2. [スキーマ] ペインの `Query` タイプを次のように変更して、新しい `allPost` クエリを追加します。

```
type Query {
    allPost(limit: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

3. 新しい `PaginationPosts` 型を追加します。

```
type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}
```

4. [Save Schema] を選択します。
5. 右側の [リゾルバー] ペインで、`allPost` タイプから新しく作成された `Query` フィールドを探し、[アタッチ] を選択します。次のコードを使用して新しいリゾルバーを作成します。

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
```

```
  const { limit = 20, nextToken } = ctx.arguments;
  return ddb.scan({ limit, nextToken });
}

export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

このリゾルバーのリクエストハンドラーには 2 つのオプション引数が必要です。

- `limit` - 1 回の呼び出しで返される項目の最大数を指定します。
- `nextToken` - 次の結果セットを取得するために使用されます (`nextToken`の値がどこから来たのかは後で説明します)。

6. リゾルバーに加えた変更を保存します。

Scan リクエストの詳細については、「Scan」リファレンスドキュメントを参照してください。

API を呼び出してすべての投稿をスキャンする

リゾルバーがセットアップされたので、 は受信`allPost`クエリを Amazon DynamoDB `Scan`オペレーションに変換する方法 AWS AppSync を理解します。ユーザーは、テーブルをスキャンしてすべてのポストを取得できるようになりました。ただし、これまで使用してきたデータはすべて削除したため、これを試す前にテーブルにデータを入力しておく必要があります。

データを追加してクエリするには

1. API で [クエリ] タブを選択します。
2. 以下のミューテーションを [クエリ] ペインに追加します。

```
mutation addPost {
  post1: addPost(id:1 author: "AUTHORNAME" title: "A series of posts, Volume 1"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post2: addPost(id:2 author: "AUTHORNAME" title: "A series of posts, Volume 2"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post3: addPost(id:3 author: "AUTHORNAME" title: "A series of posts, Volume 3"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post4: addPost(id:4 author: "AUTHORNAME" title: "A series of posts, Volume 4"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post5: addPost(id:5 author: "AUTHORNAME" title: "A series of posts, Volume 5"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
```

```
  post6: addPost(id:6 author: "AUTHORNAME" title: "A series of posts, Volume 6"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post7: addPost(id:7 author: "AUTHORNAME" title: "A series of posts, Volume 7"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post8: addPost(id:8 author: "AUTHORNAME" title: "A series of posts, Volume 8"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post9: addPost(id:9 author: "AUTHORNAME" title: "A series of posts, Volume 9"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
}
```

3. [実行] (オレンジ色の再生ボタン) を選択します。
4. では、テーブルをスキャンして、一度に 5 個の結果を返しましょう。以下のクエリをクエリペインに貼り付けます。

```
query allPost {
  allPost(limit: 5) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

5. [実行] (オレンジ色の再生ボタン) を選択し、[allPost] を選択します。

   最初の 5 個のポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        },
        {
          "id": "6",
```

```
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        }
      ],
      "nextToken": "<token>"
    }
  }
}
```

6. 5 つの結果と nextToken を取得しました。このトークンを使用して、次の結果セットを取得できます。前回の結果セットからの allPost を含めるように、nextToken クエリを更新します。

```
query allPost {
  allPost(
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
      author
    }
    nextToken
  }
}
```

7. [実行] (オレンジ色の再生ボタン) を選択し、[allPost] を選択します。

   残りの 4 個のポストが、クエリペインの右側にある結果ペインに表示されます。9 個のポストのすべてをページ分割して、ポストは残っていないため、この結果セットに nextToken はありません。これは次のように表示されます。

```
{
  "data": {
    "allPost": {
      "posts": [
        {
```

```
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        }
      ],
      "nextToken": null
    }
  }
}
```

「allPostsByAuthor」リゾルバー (Amazon DynamoDB クエリ) のセットアップ

Amazon DynamoDB ですべてのポストをスキャンだけでなく、特定の作成者が作成したポストを取得するクエリを Amazon DynamoDB に対して実行することもできます。前の手順で作成した Amazon DynamoDB テーブルには、既に `author-index` という `GlobalSecondaryIndex` があるため、Amazon DynamoDB の `Query` オペレーションでそれを使用して、特定の作成者が作成したすべてのポストを取得できます

クエリを追加するには

1. API で [スキーマ] タブを選択します。
2. [スキーマ] ペインの `Query` タイプを次のように変更して、新しい `allPostsByAuthor` クエリを追加します。

```
type Query {
    allPostsByAuthor(author: String!, limit: Int, nextToken: String): PaginatedPosts!
    allPost(limit: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
```

```
}
```

このクエリでは、allPost クエリで使用したのと同じ PaginatedPosts 型を使用していることに注意してください。

3. [Save Schema] を選択します。
4. 右側のリゾルバーペインで、allPostsByAuthor 型の新しく作成された Query フィールドを見つけて、[アタッチ] を選択します。以下のスニペットを使用してリゾルバーを作成します。

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { limit = 20, nextToken, author } = ctx.arguments;
  return ddb.query({
    index: 'author-index',
    query: { author: { eq: author } },
    limit,
    nextToken,
  });
}

export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

allPost リゾルバーと同様に、このリゾルバーには 2 つのオプション引数があります。

- limit - 1 回の呼び出しで返される項目の最大数を指定します。
- nextToken - 次の結果セットを取得します (nextTokenの値は前回の呼び出しから取得できます)。

5. リゾルバーに加えた変更を保存します。

Query リクエストの詳細については、「クエリ」リファレンスドキュメントを参照してください。

特定の作成者によるすべてのポストをクエリする API の呼び出し

リゾルバーがセットアップされたので、 はauthor-indexインデックスに対して受信allPostsByAuthorミューテーションを DynamoDB Queryオペレーションに変換する方法 AWS

AppSync を理解します。ユーザーは、テーブルをクエリして、特定の作成者によるポストをすべて取得できます。

ただし、これまで使用していたポストはすべて同じ作成者だったため、それを行う前に、テーブルに投稿を追加しておきましょう。

データとクエリを追加するには

1. API で [クエリ] タブを選択します。
2. 以下のミューテーションを [クエリ] ペインに追加します。

```
mutation addPost {
  post1: addPost(id:10 author: "Nadia" title: "The cutest dog in the world" content:
 "So cute. So very, very cute." url: "https://aws.amazon.com/appsync/" ) { author,
 title }
  post2: addPost(id:11 author: "Nadia" title: "Did you know...?" content: "AppSync
 works offline?" url: "https://aws.amazon.com/appsync/" ) { author, title }
  post3: addPost(id:12 author: "Steve" title: "I like GraphQL" content: "It's great"
 url: "https://aws.amazon.com/appsync/" ) { author, title }
}
```

3. [実行] (オレンジ色の再生ボタン) を選択し、[addPost] を選択します。
4. では、Nadia が作成したすべてのポストを返すクエリを実行しましょう。以下のクエリをクエリペインに貼り付けます。

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

5. [実行] (オレンジ色の再生ボタン) を選択し、[allPostsByAuthor] を選択します。Nadia が作成したすべてのポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
```

```
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

6. Query でのページ分割は Scan とまったく同じように動作します。例えば、AUTHORNAME によるすべてのポストを検索して、一度に 5 個ずつ取得します。
7. 以下のクエリをクエリペインに貼り付けます。

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

8. [実行] (オレンジ色の再生ボタン) を選択し、[allPostsByAuthor] を選択します。AUTHORNAME が作成したすべてのポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
```

```
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        }
      ],
      "nextToken": "<token>"
    }
  }
}
```

9. 次のように、nextToken 引数を、前回のクエリで返された値に更新します。

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

10[実行] (オレンジ色の再生ボタン) を選択し、[allPostsByAuthor] を選択します。AUTHORNAME が作成した残りのポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        },
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        }
      ],
      "nextToken": null
    }
  }
}
```

セット型の使用

ここまでの Post 型は、フラットなキーと値のオブジェクトでした。 リゾルバーを使用して、セット型、リスト型、マップ型などの複雑なオブジェクトをモデル化することもできます。Post 型を更新して、タグを含めましょう。1 つのポストには、DynamoDB に文字列として保存されているタグを 0 個以上付けることができます。タグを追加および削除するミューテーションと、特定のタグが付いているポストをスキャンする新しいクエリもセットアップします。

データを設定するには

1. API で [スキーマ] タブを選択します。

2. [スキーマ] ペインの Post タイプを次のように変更して、新しい tags フィールドを追加します。

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
}
```

3. [スキーマ] ペインの Query タイプを次のように変更して、新しい allPostsByTag クエリを追加します。

```
type Query {
  allPostsByTag(tag: String!, limit: Int, nextToken: String): PaginatedPosts!
  allPostsByAuthor(author: String!, limit: Int, nextToken: String): PaginatedPosts!
  allPost(limit: Int, nextToken: String): PaginatedPosts!
  getPost(id: ID): Post
}
```

4. [スキーマ] ペインの Mutation タイプを変更して、新しい addTag と removeTag ミューテーションを次のように追加します。

```
type Mutation {
  addTag(id: ID!, tag: String!): Post
  removeTag(id: ID!, tag: String!): Post
  deletePost(id: ID!, expectedVersion: Int): Post
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
```

```
    title: String!,
    content: String!,
    url: String!
  ): Post!
}
```

5. [Save Schema] を選択します。
6. 右側のリゾルバーペインで、allPostsByTag 型の新しく作成された Query フィールドを見つけて、[アタッチ] を選択します。以下のスニペットを使用してリゾルバーを作成します。

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { limit = 20, nextToken, tag } = ctx.arguments;
  return ddb.scan({ limit, nextToken, filter: { tags: { contains: tag } } });
}

export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

7. リゾルバーに加えた変更を保存します。
8. 次に、以下のスニペットを使ってMutation フィールド addTag に対して同じ操作を行います。

> **Note**
>
> DynamoDB ユーティリティは現在、セットオペレーションをサポートしていませんが、リクエストを自分で作成してセットを操作することはできます。

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { id, tag } = ctx.arguments
 const expressionValues = util.dynamodb.toMapValues({ ':plusOne': 1 })
 expressionValues[':tags'] = util.dynamodb.toStringSet([tag])

 return {
  operation: 'UpdateItem',
  key: util.dynamodb.toMapValues({ id }),
```

```
  update: {
   expression: `ADD tags :tags, version :plusOne`,
   expressionValues,
  },
 }
}

export const response = (ctx) => ctx.result
```

9. リゾルバーに加えた変更を保存します。

10以下のスニペットを使用して、これを Mutation フィールドでもう一度繰り返します。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
   const { id, tag } = ctx.arguments;
   const expressionValues = util.dynamodb.toMapValues({ ':plusOne': 1 });
   expressionValues[':tags'] = util.dynamodb.toStringSet([tag]);

   return {
     operation: 'UpdateItem',
     key: util.dynamodb.toMapValues({ id }),
     update: {
       expression: `DELETE tags :tags ADD version :plusOne`,
       expressionValues,
     },
   };
 }

 export const response = (ctx) => ctx.resultexport
```

11.リゾルバーに加えた変更を保存します。

タグを操作する API の呼び出し

リゾルバーをセットアップしたので、 は受信 addTag、、removeTagおよび allPostsByTagリクエストを DynamoDB UpdateItemおよび Scanオペレーションに変換する方法 AWS AppSync を理解します。それを試すには、前のステップで作成したポストのいずれかを選択します。例えば、Nadia が作成したポストを使用しましょう。

タグを使用するには

1. API で [クエリ] タブを選択します。
2. 以下のクエリをクエリペインに貼り付けます。

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

3. [実行] (オレンジ色の再生ボタン) を選択し、[allPostsByAuthor] を選択します。
4. Nadia のすべてのポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

5. タイトルが「The cutest dog in the world」の投稿を使用しましょう。id は後で使用するため書き留めておきます。では、dog タグを追加してみましょう。
6. 以下のミューテーションを [クエリ] ペインに貼り付けます。また、id 引数を、前にメモしておいた値に更新する必要があります。

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

7. [実行] (オレンジ色の再生ボタン) を選択し、[addTag] を選択します。ポストが新しいタグで更新されています。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

8. タグを追加することができます。puppy に変更するように、tag 引数を更新します。

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

9. [実行] (オレンジ色の再生ボタン) を選択し、[addTag] を選択します。投稿が新しいタグで更新されています。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
```

```
        "dog",
        "puppy"
      ]
    }
  }
}
```

10.タグを削除することもできます。以下のミューテーションを [クエリ] ペインに貼り付けます。また、id 引数を、前にメモしておいた値に更新する必要があります。

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

11.[実行] (オレンジ色の再生ボタン) を選択し、[removeTag] を選択します。ポストが更新され、puppy タグが削除されています。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

12.タグが付いているすべての投稿を検索することもできます。以下のクエリをクエリペインに貼り付けます。

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
```

```
    nextToken
  }
}
```

13[実行] (オレンジ色の再生ボタン) を選択し、[allPostsByTag] を選択します。次のように dog タグが付いているすべての投稿が返されます。

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
            "dog",
            "puppy"
          ]
        }
      ],
      "nextToken": null
    }
  }
}
```

結論

このチュートリアルでは、 AWS AppSync と GraphQL を使用して Post オブジェクトを操作できる API を構築しました。

クリーンアップするには、 コンソールから AWS AppSync GraphQL API を削除できます。

DynamoDB テーブルに関連付けられているロールを削除するには、[データソース] テーブルで [データソース]を選択し、[編集] をクリックします。[既存のロールを作成または使用する] の下にあるロールの値を書き留めます。IAM コンソールに移動して、ロールを削除します。

DynamoDB テーブルを削除するには、データソースリスト内のテーブルの名前をクリックします。これにより、DynamoDB コンソールに移動して、テーブルを削除できます。

での AWS Lambda リゾルバーの使用 AWS AppSync

AWS AppSync AWS Lambda で を使用すると、GraphQL フィールドを解決できます。例えば、GraphQL クエリが Amazon Relational Database Service (Amazon RDS) インスタンスを呼び出し、GraphQL のミューテーションを Amazon Kinesis ストリームに書き込むことができます。このセクションでは、GraphQL フィールド処理の呼び出しに応じてビジネスロジックを実行する Lambda 関数を記述する方法について説明します。

Lambda 関数を作成する

以下の例は、Node.js (runtime: Node.js 18.x) に記述された、ブログ投稿アプリケーションの一部としてブログ投稿に関するさまざまな処理を実行する Lambda 関数を示しています。コードは、.mis 拡張子の付いたファイル名で保存する必要があることに注意してください。

```
export const handler = async (event) => {
console.log('Received event {}', JSON.stringify(event, 3))

  const posts = {
1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
 content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
 AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
 content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
 '10', },
    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
 ups: null, downs: null },
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }
```

```
  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
case 'getPost':
      return posts[event.arguments.id]
    case 'allPosts':
      return Object.values(posts)
    case 'addPost':
      // return the arguments back
return event.arguments
    case 'addPostErrorWithData':
      result = posts[event.arguments.id]
      // attached additional error information to the post
      result.errorMessage = 'Error with the mutation, data has changed'
      result.errorType = 'MUTATION_ERROR'
return result
    case 'relatedPosts':
      return relatedPosts[event.source.id]
    default:
      throw new Error('Unknown field, unable to resolve ' + event.field)
  }
}
```

この Lambda 関数は、ID による投稿の取得、投稿の追加、投稿のリストの取得、および指定した投稿に関連する投稿の取得を処理します。

> **Note**
>
> `event.field` の `switch` ステートメントにより、Lambda 関数は現在解決しているフィールドを確認することができます。

AWS マネジメントコンソールを使用して、この Lambda 関数を作成します。

Lambda のデータソースを設定する

Lambda 関数を作成した後、 AWS AppSync コンソールで GraphQL API に移動し、[データソース] タブを選択します。

[データソースの作成] を選択し、[データソース名]としてわかりやすい名前 (**Lambda** など) を入力してから、[データソースタイプ] に対して [AWS Lambda 関数]を選択します。[リージョン] で、関数と

同じリージョンを選択します。[関数 ARN] で、Lambda 関数の Amazon リソースネーム (ARN)を選択します。

Lambda 関数を選択したら、新しい AWS Identity and Access Management (IAM) ロール (AWS AppSync が適切なアクセス許可を割り当てるロール) を作成するか、次のインラインポリシーを持つ既存のロールを選択できます。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/
LAMBDA_FUNCTION"
        }
    ]
}
```

また、次のように IAM ロールの AWS AppSync との信頼関係を設定する必要があります。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "appsync.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

GraphQL スキーマを作成する

データソースが Lambda 関数に接続されたので、GraphQL スキーマを作成します。

AWS AppSync コンソールのスキーマエディタから、スキーマが次のスキーマと一致していることを確認します。

```
schema {
    query: Query
    mutation: Mutation
}
type Query {
    getPost(id:ID!): Post
    allPosts: [Post]
}
type Mutation {
    addPost(id: ID!, author: String!, title: String, content: String, url: String):
 Post!
}
type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    relatedPosts: [Post]
}
```

リゾルバーを設定する

Lambda のデータソースと有効な GraphQL スキーマが登録されたので、リゾルバーを使用して GraphQL フィールドを Lambda のデータソースに接続することができます。

AWS AppSync JavaScript (APPSYNC_JS) ランタイムを使用して Lambda 関数を操作するリゾルバーを作成します。JavaScript を使用した AWS AppSync リゾルバーと関数の記述の詳細については、「リゾルバーと関数の JavaScript ランタイム機能」を参照してください。

Lambda のマッピングテンプレートの詳細については、「Lambda 向け JavaScript のリゾルバー関数リファレンス」を参照してください。

このステップでは、getPost(id:ID!): Post、allPosts: [Post]、addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!、および Post.relatedPosts: [Post] の各フィールドで Lambda 関数にリゾルバーをアタッチします。 AWS AppSync コンソールのスキーマエディタのリゾルバーペインで、getPost(id:ID!): Postフィールドの横にあるアタッチを選択します。Lambda データソースを選択します。そして、次のコードを入力します。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  return ctx.result;
}
```

このリゾルバーコードは、ソースオブジェクトを呼び出すときに、フィールド名、引数のリスト、ソースオブジェクトに関するコンテキストを Lambda 関数に渡します。[保存] を選択します。

最初のリゾルバーが正常に追加されました。残りのフィールドについてこの操作を繰り返します。

GraphQL API をテストする

これで Lambda 関数が GraphQL リゾルバーに接続されたので、コンソールまたはクライアントアプリケーションを使用してミューテーションまたはクエリが実行できます。

AWS AppSync コンソールの左側でクエリを選択し、次のコードを貼り付けます。

addPost ミューテーション

```
mutation AddPost {
    addPost(
        id: 6
        author: "Author6"
        title: "Sixth book"
        url: "https://www.amazon.com/"
        content: "This is the book is a tutorial for using GraphQL with AWS AppSync."
```

```
    ) {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

getPost クエリ

```
query GetPost {
    getPost(id: "2") {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

allPosts クエリ

```
query AllPosts {
    allPosts {
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {
            id
            title
        }
    }
}
```

エラーを返す

指定したフィールドの解決でエラーが発生する場合があります。 AWS AppSync を使用すると、次のソースからエラーを生成できます。

- リゾルバーレスポンスハンドラー
- Lambda function

リゾルバーレスポンスハンドラーから

意図的なエラーを発生させるには、`util.error` ユーティリティメソッドを使用できます。これには引数として、`errorMessage`、`errorType`、および必要に応じて `data` の各値を使用します。`data` は、エラーの発生時にクライアントに追加のデータを返す場合に利用できます。`data` オブジェクトは GraphQL の最終レスポンスで `errors` に追加されます。

次の例は、`Post.relatedPosts: [Post]` リゾルバーのレスポンスハンドラーでこれを使用する方法を示しています。

```
// the Post.relatedPosts response handler
export function response(ctx) {
    util.error("Failed to fetch relatedPosts", "LambdaFailure", ctx.result)
    return ctx.result;
}
```

これにより、以下のような GraphQL レスポンスが生成されます。

```
{
    "data": {
        "allPosts": [
            {
                "id": "2",
                "title": "Second book",
                "relatedPosts": null
            },
            ...
        ]
    },
    "errors": [
        {
            "path": [
```

```
                "allPosts",
                0,
                "relatedPosts"
            ],
            "errorType": "LambdaFailure",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ],
            "message": "Failed to fetch relatedPosts",
            "data": [
                {
                  "id": "2",
                  "title": "Second book"
                },
                {
                  "id": "1",
                  "title": "First book"
                }
            ]
        }
    ]
}
```

この場合、エラーおよび `errorMessage`、`errorType`、`data` が `data.errors[0]` オブジェクトに存在するため、`allPosts[0].relatedPosts` は null になります。

Lambda 関数から

AWS AppSync は、Lambda 関数がスローするエラーも理解します。Lambda プログラミングモデルを使用し、処理されるエラーを発生させることができます。Lambda 関数からエラーがスローされた場合、 AWS AppSync は現在のフィールドの解決に失敗します。Lambda から返されたエラーメッセージのみがレスポンスに設定されます。また現在のところ、Lambda 関数からエラーを発生させて、クライアントにエラー関連以外のデータを渡すことはできません。

> **Note**
>
> 注意 : Lambda 関数が処理されないエラーを発生させた場合、 AWS AppSync は によって設定されたエラーメッセージを使用します。

以下の Lambda 関数はエラーを発生させます。

```
export const handler = async (event) => {
  console.log('Received event {}', JSON.stringify(event, 3))
  throw new Error('I always fail.')
}
```

エラーはレスポンスハンドラーで受信されます。util.appendError でレスポンスにエラーを追加することで、GraphQL レスポンスで返すことができます。これを行うには、 AWS AppSync 関数レスポンスハンドラーを次のように変更します。

```
// the lambdaInvoke response handler
export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  }
  return result;
}
```

これにより、以下のような GraphQL レスポンスが返されます。

```
{
  "data": {
    "allPosts": null
  },
  "errors": [
    {
      "path": [
        "allPosts"
      ],
      "data": null,
      "errorType": "Lambda:Unhandled",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "I fail. always"
```

```
    }
  ]
}
```

高度なユースケース: バッチ処理

この例の Lambda 関数には、指定した投稿に関連する投稿のリストを返す relatedPosts フィールドが含まれています。この例のクエリでは、Lambda 関数からの allPosts フィールドの呼び出しにより 5 件の投稿が返されます。返された各投稿に対して relatedPosts の解決も指定しているので、relatedPosts フィールドの処理が続けて 5 回呼び出されます。

```
query {
    allPosts {   // 1 Lambda invocation - yields 5 Posts
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {   // 5 Lambda invocations - each yields 5 posts
            id
            title
        }
    }
}
```

この例では大して意味がないように思われますが、こうした余分な取得処理が積み重なることで、アプリケーションに急速に害をなす可能性があります。

同じクエリで返された関連する Posts について、再度 relatedPosts を取得すると、呼び出しの数は大幅に増加します。

```
query {
    allPosts {   // 1 Lambda invocation - yields 5 Posts
        id
        author
        title
        content
        url
        ups
        downs
```

```
        relatedPosts {   // 5 Lambda invocations - each yield 5 posts = 5 x 5 Posts
            id
            title
            relatedPosts {  // 5 x 5 Lambda invocations - each yield 5 posts = 25 x 5
 Posts
                id
                title
                author
            }
        }
    }
}
```

この比較的単純なクエリでは、 AWS AppSync は Lambda 関数を 1 + 5 + 25 = 31 回呼び出します。

これはよくある課題であり、N+1 問題と呼ばれます (この例では、N = 5)。これによりアプリケーションのレイテンシーとコストが増大します。

この問題を解決する 1 つの方法は、同様なフィールドのリゾルバーリクエストを同時にバッチ処理することです。この例では、Lambda 関数は指定された 1 つの投稿に関連する投稿のリストを解決するのではなく、指定された一連の投稿に関連する投稿のリストを解決できます。

これを実証するために、relatedPosts に対するバッチ処理を行うようにリゾルバーを更新してみましょう。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: ctx.info.fieldName === 'relatedPosts' ? 'BatchInvoke' : 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  }
  return result;
}
```

このコードでは、解決対象のfieldName が relatedPosts である場合に、オペレーションが Invoke から BatchInvoke に変更されます。次に、[バッチ処理の設定] セクションで関数のバッチ処理を有効にします。最大バッチサイズを 5 に設定します。[保存] を選択します。

この変更により、Lambda 関数は relatedPosts の解決時に入力として以下を受け取ります。

```
[
    {
        "field": "relatedPosts",
        "source": {
            "id": 1
        }
    },
    {
        "field": "relatedPosts",
        "source": {
            "id": 2
        }
    },
    ...
]
```

リクエストで BatchInvoke が指定されている場合、Lambda 関数はリクエストのリストを受け取り、結果のリストを返します。

具体的には、 AWS AppSync がそれに応じて結果を一致できるように、結果のリストがリクエストペイロードエントリのサイズと順序と一致する必要があります。

このバッチ処理の例では、Lambda 関数は次のように一連の結果を返します。

```
[
    [{"id":"2","title":"Second book"}, {"id":"3","title":"Third book"}],   //
 relatedPosts for id=1
    [{"id":"3","title":"Third book"}]                                      //
 relatedPosts for id=2
]
```

Lambda コードを更新して、relatedPosts に対するバッチ処理を行うことができます。

```
export const handler = async (event) => {
  console.log('Received event {}', JSON.stringify(event, 3))
  //throw new Error('I fail. always')
```

```
  const posts = {
    1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
 content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
 AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
 content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
 '10', },
    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
 ups: null, downs: null },
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
    1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }

  if (!event.field && event.length){
    console.log(`Got a BatchInvoke Request. The payload has ${event.length} items to
 resolve.`);
    return event.map(e => relatedPosts[e.source.id])
  }

  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
    case 'getPost':
      return posts[event.arguments.id]
    case 'allPosts':
      return Object.values(posts)
    case 'addPost':
      // return the arguments back
      return event.arguments
    case 'addPostErrorWithData':
```

```
      result = posts[event.arguments.id]
      // attached additional error information to the post
      result.errorMessage = 'Error with the mutation, data has changed'
      result.errorType = 'MUTATION_ERROR'
      return result
    case 'relatedPosts':
      return relatedPosts[event.source.id]
    default:
      throw new Error('Unknown field, unable to resolve ' + event.field)
  }
}
```

個々のエラーを返す

Lambda 関数から単一のエラーを返せることや、レスポンスハンドラーから 1 つのエラーを生成できることは以前の例で説明しました。バッチ処理を呼び出した場合、Lambda 関数からエラーが発生すると、バッチ処理全体が失敗としてフラグ付けされます。これは、データストアとの接続が切れた場合など、回復不可能なエラーが発生するシナリオでは問題ないかもしれません。ここでは、バッチ処理の一部が成功し、他が失敗した場合、エラーと有効なデータの両方を返すことができます。 AWS AppSync では、バッチ処理のレスポンスがバッチの元のサイズと一致する要素のリストとなるように要求されるため、エラーから有効なデータが識別できるようなデータ構造を定義する必要があります。

例えば、関連する一連の投稿が返されることを Lambda 関数が期待している場合には、data、errorMessage、errorType の各フィールドを任意に含む Response オブジェクトのリストを返します。errorMessage フィールドが存在する場合は、エラーが発生したことを意味します。

次のコードは Lambda 関数の更新方法を示しています。

```
export const handler = async (event) => {
console.log('Received event {}', JSON.stringify(event, 3))
  // throw new Error('I fail. always')
const posts = {
1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
 content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
 AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
 content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
 '10', },
    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
 ups: null, downs: null },
```

```
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }

  if (!event.field && event.length){
console.log(`Got a BatchInvoke Request. The payload has ${event.length} items to
 resolve.`);
    return event.map(e => {
// return an error for post 2
if (e.source.id === '2') {
return { 'data': null, 'errorMessage': 'Error Happened', 'errorType': 'ERROR' }
      }
      return {data: relatedPosts[e.source.id]}
      })
  }

  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
case 'getPost':
      return posts[event.arguments.id]
    case 'allPosts':
      return Object.values(posts)
    case 'addPost':
      // return the arguments back
return event.arguments
    case 'addPostErrorWithData':
      result = posts[event.arguments.id]
      // attached additional error information to the post
      result.errorMessage = 'Error with the mutation, data has changed'
      result.errorType = 'MUTATION_ERROR'
```

```
return result
    case 'relatedPosts':
      return relatedPosts[event.source.id]
    default:
      throw new Error('Unknown field, unable to resolve ' + event.field)
  }
}
```

relatedPosts リゾルバーコードを更新します。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: ctx.info.fieldName === 'relatedPosts' ? 'BatchInvoke' : 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  } else if (result.errorMessage) {
    util.appendError(result.errorMessage, result.errorType, result.data)
  } else if (ctx.info.fieldName === 'relatedPosts') {
      return result.data
  } else {
      return result
  }
}
```

レスポンスハンドラーは、Invoke オペレーションで Lambda 関数から返されるエラーをチェックし、BatchInvoke オペレーションの個々の項目について返されたエラーをチェックしてから、最後に fieldName をチェックするようになりました。relatedPostsに対して、関数は result.data を返します。その他のすべてのフィールドでは、関数は result を返すだけです。例えば、以下のクエリを参照してください。

```
query AllPosts {
  allPosts {
    id
```

```
    title
    content
    url
    ups
    downs
    relatedPosts {
      id
    }
    author
  }
}
```

このクエリでは、次のような GraphQL レスポンスが返されます。

```
{
  "data": {
    "allPosts": [
      {
        "id": "1",
        "relatedPosts": [
          {
            "id": "4"
          }
        ]
      },
      {
        "id": "2",
        "relatedPosts": null
      },
      {
        "id": "3",
        "relatedPosts": [
          {
            "id": "2"
          },
          {
            "id": "1"
          }
        ]
      },
      {
        "id": "4",
        "relatedPosts": [
```

```
          {
            "id": "2"
          },
          {
            "id": "1"
          }
        ]
      },
      {
        "id": "5",
        "relatedPosts": []
      }
    ]
  },
  "errors": [
    {
      "path": [
        "allPosts",
        1,
        "relatedPosts"
      ],
      "data": null,
      "errorType": "ERROR",
      "errorInfo": null,
      "locations": [
        {
          "line": 4,
          "column": 5,
          "sourceName": null
        }
      ],
      "message": "Error Happened"
    }
  ]
}
```

最大バッチサイズの設定

リゾルバーで最大バッチサイズを設定するには、 AWS Command Line Interface () で次のコマンドを使用しますAWS CLI。

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
 relatedPosts \
```

```
--code "<code-goes-here>" \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--data-source-name "<lambda-datasource>" \
--max-batch-size X
```

> **Note**
>
> リクエストマッピングテンプレートを提供するときは、バッチ処理を使用する `BatchInvoke` オペレーションを使用する必要があります。

AWS AppSync でのローカルリゾルバーの使用

AWS AppSync では、サポートされているデータソース (AWS Lambda、Amazon DynamoDB、または Amazon OpenSearch Service) を使用してさまざまなオペレーションを実行できます。ただし、特定のシナリオでは、サポートされているデータソースに対する呼び出しの必要がないことがあります。

そのような場合は、ローカルリゾルバーが役立ちます。リモートのデータソースを呼び出すのではなく、ローカルリゾルバーはリクエストハンドラーの結果をレスポンスハンドラーに転送します。フィールド解決は AWS AppSync を離れません。

ローカルリゾルバーは、さまざまな状況で役立ちます。特によく使用されるのは、データソースの呼び出しをトリガーせずに通知を発行する場合です。このユースケースを実証するために、ユーザーがメッセージをパブリッシュしたりサブスクライブしたりできる pub/sub アプリケーションを作成してみましょう。この例では、サブスクリプションを活用します。サブスクリプションに慣れていない場合は、「リアルタイムデータ」チュートリアルを参照してください。

Pub/Sub アプリの作成

まず、[最初から設計] オプションを選択し、GraphQL API の作成時にオプションの詳細を設定して、空白の GraphQL API を作成します。

Pub/sub アプリケーションでは、クライアントはメッセージをサブスクライブしたりパブリッシュしたりできます。パブリッシュされた各メッセージには名前とデータが含まれます。これをスキーマに追加してください。

```
type Channel {
 name: String!
```

```
 data: AWSJSON!
}

type Mutation {
 publish(name: String!, data: AWSJSON!): Channel
}

type Query {
 getChannel: Channel
}

type Subscription {
 subscribe(name: String!): Channel
  @aws_subscribe(mutations: ["publish"])
}
```

次に、リゾルバーを Mutation.publish フィールドにアタッチしましょう。「スキーマ」ペインの横にある「リゾルバー」ペインで、Mutation タイプを探し、publish(...): Channel フィールドを探して、「アタッチ」をクリックします。

None データソースを作成して、PageDataSource という名前を付けます。リゾルバーにアタッチします。

次のスニペットを使用してリゾルバーの実装を追加します:

```
export function request(ctx) {
  return { payload: ctx.args };
}

export function response(ctx) {
  return ctx.result;
}
```

必ずリゾルバーを作成し、変更を保存してください。

ページの送信およびサブスクライブ

クライアントがメッセージを受信するためには、まずは受信箱をサブスクライブする必要があります。

[クエリ] ペインで SubscribeToData のサブスクリプションを実行します。

```
subscription SubscribeToData {
    subscribe(name:"channel") {
        name
        data
    }
}
```

サブスクライバーは、publish ミューテーションが呼び出されるたびにメッセージを受信しますが、メッセージが channel サブスクリプションに送信されたときだけです。クエリペインでこれを試してみましょう。サブスクリプションがまだコンソールで実行されている間に、別のコンソールを開き、クエリペインで次のリクエストを実行します。

> **Note**
>
> この例では、有効な JSON 文字列を使用しています。

```
mutation PublishData {
    publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
        data
        name
    }
}
```

結果は以下のようになります。

```
{
  "data": {
    "publish": {
      "data": "{\"msg\":\"hello world!\"}",
      "name": "channel"
    }
  }
}
```

ここまで、 AWS AppSync サービスを離れることなくメッセージを発行して受信することで、ローカルリゾルバーの使用をデモンストレーションしました。

AWS AppSyncでの GraphQL リゾルバーの組み合わせ

GraphQL スキーマのリゾルバーとフィールドには、非常に柔軟性の高い 1 対 1 の関係があります。データソースはスキーマとは独立してリゾルバーに設定されるため、ニーズに応じてスキーマ上で組み合わせやマッチングを行い、さまざまなデータソースを使用して GraphQL 型の解決や操作が行うことができます。

次のシナリオでは、スキーマ内のデータソースを組み合わせやマッチングを行う方法を紹介します。開始する前に、、Amazon DynamoDB AWS Lambda、および Amazon OpenSearch Service のデータソースとリゾルバーの設定について理解しておく必要があります。

スキーマの例

次のスキーマは `Post` タイプで、それぞれ 3 つの `Query` と `Mutation` オペレーションがあります。

```
type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
}

type Query {
    allPost: [Post]
    getPost(id: ID!): Post
    searchPosts: [Post]
}

type Mutation {
    addPost(
        id: ID!,
        author: String!,
        title: String,
        content: String,
        url: String
    ): Post
    updatePost(
        id: ID!,
```

```
        author: String!,
        title: String,
        content: String,
        url: String,
        ups: Int!,
        downs: Int!,
        expectedVersion: Int!
    ): Post
    deletePost(id: ID!): Post
}
```

この例では、それぞれにデータソースが必要な、合計で 6 つのリゾルバーを使用します。この問題を解決する 1 つの方法は、Posts と呼ばれる 1 つの Amazon DynamoDB テーブルにこれらをフックすることです。このテーブルでは、AllPost フィールドがスキャンを実行し、searchPosts フィールドがクエリを実行します (「DynamoDB 用の JavaScript リゾルバー関数リファレンス」を参照)。ただし、Amazon DynamoDB に限ったことではありません。Lambda や OpenSearch Service など、ビジネス要件を満たすために存在してします。

リゾルバーを使用してデータを変更する

AWS AppSync データソースによって直接サポートされていないサードパーティーデータベースから結果を返す必要がある場合があります。また、データを API クライアントに返す前に、データに複雑な変更を加える必要がある場合もあります。これは、クライアントでのタイムスタンプの相違や、後方互換性の問題の処理など、データ型の不適切なフォーマットが原因の可能性があります。この場合、 AWS Lambda 関数をデータソースとして AWS AppSync API に接続するのが適切なソリューションです。説明のために、次の例では、 AWS Lambda 関数はサードパーティーのデータストアから取得されたデータを操作します。

```
export const handler = (event, context, callback) => {
    // fetch data
    const result = fetcher()

    // apply complex business logic
    const data = transform(result)

    // return to AppSync
    return data
};
```

これは完全に有効な Lambda 関数であり、GraphQL スキーマの `AllPost` フィールドにアタッチできるため、すべての結果を返すクエリが、賛成/反対に対するランダムな番号を受け取ります。

DynamoDB と OpenSearch Service

一部のアプリケーションでは、DynamoDB に対してミューテーションまたは簡単な検索クエリを実行し、バックグラウンドプロセスでドキュメントを OpenSearch Service に転送する場合があります。その後、`searchPosts` リゾルバーを OpenSearch Service データソースに単純にアタッチし、GraphQL クエリを使用して、(DynamoDB のデータからの) 検索結果を返します。これは、キーワードやあいまいワードによる検索、地理空間検索などの高度な検索処理をアプリケーションに追加する場合に非常に役立ちます。DynamoDB からのデータ転送は、ETL プロセスを使用して、または Lambda を使用した DynamoDB からのストリームを使用して行うことができます。

これらの特定のデータソースを使い始めるには、DynamoDB と Lambda のチュートリアルを参照してください。

例えば、前のチュートリアルのスキーマを使用すると、次のミューテーションによって DynamoDB に項目が追加されます。

```
mutation addPost {
  addPost(
    id: 123
    author: "Nadia"
    title: "Our first post!"
    content: "This is our first post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

これにより、データが DynamoDB に書き込まれます。その後、Lambda を介してデータが Amazon OpenSearch Service に転送されます。ここではすべての投稿がさまざまなフィールドについて検索

できます。例えば、データが Amazon OpenSearch Service にあるため、以下のようにスペースを含む任意のテキストを使用して、author フィールドまたは content フィールドを検索できます。

```
query searchName{
    searchAuthor(name:"   Nadia   "){
        id
        title
        content
    }
}

---------- or ----------

query searchContent{
    searchContent(text:"test"){
        id
        title
        content
    }
}
```

データは DynamoDB に直接書き込まれるため、`allPost{...}` クエリと `getPost{...}` クエリを使用して、テーブルに対するリストまたは項目の検索処理が効率的に実行できます。このスタックでは、DynamoDB ストリームに対して次のコード例を使用します。

> **Note**
>
> この Python コードは一例であり、本番稼働用コードでの使用を意図したものではありません。

```
import boto3
import requests
from requests_aws4auth import AWS4Auth

region = '' # e.g. us-east-1
service = 'es'
credentials = boto3.Session().get_credentials()
awsauth = AWS4Auth(credentials.access_key, credentials.secret_key, region, service,
 session_token=credentials.token)
```

```
host = '' # the OpenSearch Service domain, e.g. https://search-mydomain.us-
west-1.es.amazonaws.com
index = 'lambda-index'
datatype = '_doc'
url = host + '/' + index + '/' + datatype + '/'

headers = { "Content-Type": "application/json" }

def handler(event, context):
    count = 0
    for record in event['Records']:
        # Get the primary key for use as the OpenSearch ID
        id = record['dynamodb']['Keys']['id']['S']

        if record['eventName'] == 'REMOVE':
            r = requests.delete(url + id, auth=awsauth)
        else:
            document = record['dynamodb']['NewImage']
            r = requests.put(url + id, auth=awsauth, json=document, headers=headers)
        count += 1
    return str(count) + ' records processed.'
```

DynamoDB ストリームを使用し、 をプライマリキーとしてこれを DynamoDB テーブルにアタッチできます。DynamoDB のソースへの変更は OpenSearch Service ドメインに転送されます。この設定の詳細については、DynamoDB Streams のドキュメントを参照してください。

AWS AppSync での Amazon OpenSearch Service リゾルバーの使用

AWS AppSync は、VPC 内に存在しない場合、独自の AWS アカウントでプロビジョニングしたドメインからの Amazon OpenSearch Service の使用をサポートします。ドメインをプロビジョニングした後、データソースを使用してそれに接続できます。この時点で、スキーマにリゾルバーを設定し、クエリ、ミューテーション、サブスクリプションなどの GraphQL 処理を実行することができます。このチュートリアルでは、いくつかの一般的な例を説明します。

詳細については、「OpenSearch 用の JavaScript リゾルバー関数リファレンス」を参照してください。

新しい OpenSearch Service ドメインを作成する

このチュートリアルを開始するには、既存の OpenSearch Service ドメインが必要です。ドメインがない場合は、次のサンプルが使用できます。OpenSearch Service ドメインが作成されるまで、AWS AppSync データソースとの統合に進むまでに最大 15 分かかる場合があります。

```
aws cloudformation create-stack --stack-name AppSyncOpenSearch \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/elasticsearch/
ESResolverCFTemplate.yaml \
--parameters ParameterKey=OSDomainName,ParameterValue=ddtestdomain
 ParameterKey=Tier,ParameterValue=development \
--capabilities CAPABILITY_NAMED_IAM
```

アカウントの US-West-2 (オレゴン) リージョンで次の AWS CloudFormation スタックを起動できます AWS 。

OpenSearch Service 用のデータソースの設定

OpenSearch Service ドメインを作成したら、 AWS AppSync GraphQL API に移動し、データソースタブを選択します。[データソースの作成] を選択して、データソースに「*oss*」などのわかりやすい名前を入力します。次に、[データソースタイプ] に [Amazon OpenSearch ドメイン] を選択し、適切なリージョンを選択します。OpenSearch Service ドメインがリストに表示されます。選択したら、新しいロールを作成するか、 AWS AppSync がロールに適切なアクセス許可を割り当てるか、次のインラインポリシーを持つ既存のロールを選択できます。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1234234",
            "Effect": "Allow",
            "Action": [
                "es:ESHttpDelete",
                "es:ESHttpHead",
                "es:ESHttpGet",
```

```
                "es:ESHttpPost",
                "es:ESHttpPut"
            ],
            "Resource": [
                "arn:aws:es:us-east-1:ACCOUNTNUMBER:domain/democluster/*"
            ]
        }
    ]
}
```

また、そのロールに対して AWS AppSync との信頼関係を設定する必要があります。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "appsync.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

さらに、OpenSearch Service ドメインには独自のアクセスポリシーがあり、Amazon Amazon OpenSearch Service コンソールから変更できます。OpenSearch Service ドメインの適切なアクションとリソースを用いて、以下のようなポリシーを追加する必要があります。プリンシパルは AWS AppSync データソースロールになることに注意してください。プリンシパルは IAM コンソールで作成できます。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:role/service-role/
APPSYNC_DATASOURCE_ROLE"
            },
            "Action": [
                "es:ESHttpDelete",
                "es:ESHttpHead",
                "es:ESHttpGet",
                "es:ESHttpPost",
                "es:ESHttpPut"
            ],
            "Resource": "arn:aws:es:us-east-1:ACCOUNTNUMBER:domain/DOMAIN_NAME/*"
        }
    ]
}
```

リゾルバーを接続する

これで、データソースが OpenSearch Service ドメインに接続されたので、リゾルバーを使用して、以下の例のようにデータソースを GraphQL スキーマに接続することができます。

```
 type Query {
   getPost(id: ID!): Post
   allPosts: [Post]
 }

 type Mutation {
   addPost(id: ID!, author: String, title: String, url: String, ups: Int, downs: Int,
 content: String): AWSJSON
 }

type Post {
  id: ID!
  author: String
  title: String
  url: String
  ups: Int
  downs: Int
  content: String
}
```

Post フィールドを含むユーザー定義の id 型があることに注意してください。次の例では、この型を OpenSearch Service ドメインに配置するプロセス (自動化可能) があることを前提としています。このプロセスでは、/post/_doc へのルートパスをマッピングします。post はインデックスです。このルートパスから、個々のドキュメントの検索、/id/post* によるワイルドカード検索、/post/_search の パスを使用した複数ドキュメントの検索が行うことができます。例えば、同じインデックス user でインデックスが付けられた User という別の型がある場合は、/user/_search の パスを使用して複数ドキュメントが検索できます。

AWS AppSync コンソールのスキーマエディタで、前述のPostsスキーマを変更してsearchPostsクエリを含めます。

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

スキーマを保存します。「リゾルバー」ペインで searchPosts を見つけ、「アタッチ」を選択します OpenSearch Service のデータソースを選択し、リゾルバーを保存します。以下のスニペットを使用してリゾルバーのコードを更新してください。

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by using an input term
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
 return {
  operation: 'GET',
  path: `/post/_search`,
  params: { body: { from: 0, size: 50 } },
 }
}

/**
 * Returns the fetched items
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
```

```
export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type)
 }
 return ctx.result.hits.hits.map((hit) => hit._source)
}
```

これは、前述のスキーマの post フィールドの下に OpenSearch Service でインデックス化されたドキュメントがあることを前提としています。データ構造が異なる場合は、更新が必要です。

検索を変更する

前述のリゾルバーリクエストハンドラーは、すべてのレコードに簡単なクエリを実行します。今、特定の筆者で検索する場合を考えます。また、その筆者を、GraphQL クエリに定義した引数にするとします。 AWS AppSync コンソールのスキーマエディタで、allPostsByAuthorクエリを追加します。

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

「リゾルバー」ペインで allPostsByAuthor を見つけて「アタッチ」を選択します。OpenSearch Service のデータソースを選択し、以下のコードを使用してください。

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
 return {
  operation: 'GET',
  path: '/post/_search',
  params: {
   body: {
    from: 0,
```

```
    size: 50,
    query: { match: { author: ctx.args.author } },
   },
  },
 }
}

/**
 * Returns the fetched items
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type)
 }
 return ctx.result.hits.hits.map((hit) => hit._source)
}
```

body には author フィールドを含む term クエリが含まれていることに注意してください。これは引数としてクライアントから渡されます。任意で、標準テキストなどの事前に入力された情報を追加することもできます。

OpenSearch Service へのデータの追加

GraphQL ミューテーションの結果、OpenSearch Service ドメインにデータの追加が必要になる場合があります。これは検索やその他の目的に非常に役立ちます。GraphQL サブスクリプションを使用してデータをリアルタイムで作成できるため、これは OpenSearch Service ドメインのデータの更新をクライアントに通知するメカニズムとして動作します。

AWS AppSync コンソールのスキーマページに戻り、addPost()ミューテーションにアタッチを選択します。OpenSearch Service のデータソースをもう一度選択し、以下のコードを使用してください。

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
```

```
 return {
  operation: 'PUT',
  path: `/post/_doc/${ctx.args.id}`,
  params: { body: ctx.args },
 }
}

/**
 * Returns the inserted post
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type)
 }
 return ctx.result
}
```

以前と同様、これはデータ構造の一例です。別のフィールド名またはインデックスがある場合は、必要に応じて path と body を更新します。この例では context.arguments の使用方法も示しています。リクエストハンドラーはctx.argsと記述することもできます。

単一のドキュメントの取得

最後に、スキーマでgetPost(id:ID)クエリを使用して個々のドキュメントを返す場合は、 AWS AppSync コンソールのスキーマエディタでこのクエリを検索し、Attach を選択します。OpenSearch Service のデータソースをもう一度選択し、以下のコードを使用してください。

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
 return {
  operation: 'GET',
  path: `/post/_doc/${ctx.args.id}`,
 }
}
```

```
/**
 * Returns the post
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type)
 }
 return ctx.result._source
}
```

クエリとミューテーションを実行する

これで、OpenSearch Service ドメインに対して GraphQL 処理が実行できます。 AWS AppSync コンソールのクエリタブに移動し、新しいレコードを追加します。

```
mutation AddPost {
    addPost (
        id:"12345"
        author: "Fred"
        title: "My first book"
        content: "This will be fun to write!"
        url: "publisher website",
        ups: 100,
        downs:20
       )
}
```

ミューテーションの結果が右側に表示されます。同様に、OpenSearch Service ドメインに対して `searchPosts` クエリが実行できます。

```
query search {
    searchPosts {
        id
        title
        author
        content
    }
}
```

ベストプラクティス

- OpenSearch Service はプライマリデータベースとしてではなく、データのクエリ発行のために使用します。「GraphQL リゾルバーを組み合わせる」で説明したように、Amazon DynamoDB と組み合わせて OpenSearch Service を使用する場合があります。
- ドメインへのアクセスを許可するには、 AWS AppSync サービスロールにクラスターへのアクセスを許可します。
- 開発中は、最小限のコストのクラスターを使用して小規模で開始し、その後、本稼働への移行時に高可用性 (HA) を備えた大規模なクラスターへと移行することができます。

AWS AppSync での DynamoDB トランザクションの実行

AWS AppSync は、1 つのリージョン内の 1 つ以上のテーブルで Amazon DynamoDB トランザクションオペレーションの使用をサポートしています。サポートされている処理は、`TransactGetItems` および `TransactWriteItems` です。 AWS AppSync でこれらの機能を使用すると、次のようなタスクを実行できます。

- 単一クエリでキーのリストを渡し、テーブルからの結果を返す
- 単一クエリで 1 つ以上のテーブルからレコードを読み取る
- トランザクション内のレコードを 1 つまたは複数のテーブルにオールオアナッシング方式で書き込みます
- 一部の条件が満たされたときにトランザクションを実行します

アクセス許可

他のリゾルバーと同様に、 AWS AppSync でデータソースを作成し、ロールを作成するか、既存のロールを使用する必要があります。トランザクションオペレーションでは DynamoDB テーブルごとに異なるアクセス許可が必要であるため、読み取りまたは書き込みアクションのために設定されたアクセス許可がロールに必要です。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:us-east-1:accountId:table/TABLENAME",
                "arn:aws:dynamodb:region:accountId:table/TABLENAME/*"
            ]
        }
    ]
}
```

> **Note**
>
> ロールは AWS AppSync のデータソースに関連付けられ、フィールドのリゾルバーはデータソースに対して呼び出されます。DynamoDB に対して取得するよう設定されたデータソースは、設定を単純にするために 1 つのテーブルのみ指定されます。したがって、単一のリゾルバーで複数のテーブルに対してトランザクションオペレーションを実行する場合、これはより高度なタスクであり、リゾルバーが利用するすべてのテーブルへのアクセスをデータソースのロールに許可する必要があります。これは、上記の IAM ポリシーの Resource フィールドで行われます。テーブルに対するトランザクション呼び出しの設定は、リゾルバーのテンプレートで行います。これについては以下で説明します。

データソース

分かりやすくするために、このチュートリアルではすべてのリゾルバーに同じデータソースを使用します。

savingAccounts および checkingAccounts という 2 つテーブルができます。両方ともパーティションキーとして `accountNumber` を transactionHistory テーブル`transactionId`と一緒に持ちます。次の CLI コマンドをと共に使用して、テーブルを作成できます。必ずリージョンを `region` に置き換えてください。

CLI を使用する

```
aws dynamodb create-table --table-name savingAccounts \
  --attribute-definitions AttributeName=accountNumber,AttributeType=S \
  --key-schema AttributeName=accountNumber,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region

aws dynamodb create-table --table-name checkingAccounts \
  --attribute-definitions AttributeName=accountNumber,AttributeType=S \
  --key-schema AttributeName=accountNumber,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region

aws dynamodb create-table --table-name transactionHistory \
  --attribute-definitions AttributeName=transactionId,AttributeType=S \
  --key-schema AttributeName=transactionId,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region
```

AWS AppSync コンソールのデータソースで、新しい DynamoDB データソースを作成し、TransactTutorial という名前を付けます。テーブルとして savingAccounts を選択します (ただし、トランザクションを使用する場合は特定のテーブルは関係ありません)。[新規] を選択して、新しいロールとデータソースを作成します。データソースの設定を確認して、生成されたロールの名前を確認できます。IAM コンソールでは、データソースがすべてのテーブルを操作できるようにするインラインポリシーを追加できます。

`region` および `accountID` をお客様のリージョンとアカウント ID に置き換えます。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
```

```
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:us-east-1:accountId:table/savingAccounts",
                "arn:aws:dynamodb:region:accountId:table/savingAccounts/*",
                "arn:aws:dynamodb:us-east-1:accountId:table/checkingAccounts",
                "arn:aws:dynamodb:region:accountId:table/checkingAccounts/*",
                "arn:aws:dynamodb:us-east-1:accountId:table/transactionHistory",
                "arn:aws:dynamodb:region:accountId:table/transactionHistory/*"
            ]
        }
    ]
}
```

トランザクション

この例では、コンテキストは従来の銀行取引で、次の目的で TransactWriteItems を使用します。

- 普通預金から当座預金への振替
- 取引ごとの新しい取引レコードの生成

次に、TransactGetItems を使用して、普通預金と当座預金の詳細を取得します。

> **Warning**
>
> TransactWriteItems は、競合の検出と解決とともに使用する場合はサポートされていません。これらの設定は、起こり得るエラーを防ぐために無効にする必要があります。

次のように GraphQL スキーマを定義します。

```
type SavingAccount {
    accountNumber: String!
    username: String
    balance: Float
}

type CheckingAccount {
    accountNumber: String!
    username: String
```

```
    balance: Float
}

type TransactionHistory {
    transactionId: ID!
    from: String
    to: String
    amount: Float
}

type TransactionResult {
    savingAccounts: [SavingAccount]
    checkingAccounts: [CheckingAccount]
    transactionHistory: [TransactionHistory]
}

input SavingAccountInput {
    accountNumber: String!
    username: String
    balance: Float
}

input CheckingAccountInput {
    accountNumber: String!
    username: String
    balance: Float
}

input TransactionInput {
    savingAccountNumber: String!
    checkingAccountNumber: String!
    amount: Float!
}

type Query {
    getAccounts(savingAccountNumbers: [String], checkingAccountNumbers: [String]):
 TransactionResult
}

type Mutation {
    populateAccounts(savingAccounts: [SavingAccountInput], checkingAccounts:
 [CheckingAccountInput]): TransactionResult
    transferMoney(transactions: [TransactionInput]): TransactionResult
```

```
}
```

TransactWriteItems - 口座の入力

口座間で振替を行うには、テーブルに詳細を入力する必要があります。これを実行するには、GraphQL オペレーション `Mutation.populateAccounts` を使用します。

[スキーマ] セクションで、`Mutation.populateAccounts` オペレーションの横にある [アタッチ] をクリックします。`TransactTutorial` データソースを選択し、[作成] を選択します。

ここで、以下のコードを使用します。

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { savingAccounts, checkingAccounts } = ctx.args

 const savings = savingAccounts.map(({ accountNumber, ...rest }) => {
  return {
   table: 'savingAccounts',
   operation: 'PutItem',
   key: util.dynamodb.toMapValues({ accountNumber }),
   attributeValues: util.dynamodb.toMapValues(rest),
  }
 })

 const checkings = checkingAccounts.map(({ accountNumber, ...rest }) => {
  return {
   table: 'checkingAccounts',
   operation: 'PutItem',
   key: util.dynamodb.toMapValues({ accountNumber }),
   attributeValues: util.dynamodb.toMapValues(rest),
  }
 })
 return {
  version: '2018-05-29',
  operation: 'TransactWriteItems',
  transactItems: [...savings, ...checkings],
 }
}

export function response(ctx) {
 if (ctx.error) {
```

```
  util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
 }
 const { savingAccounts: sInput, checkingAccounts: cInput } = ctx.args
 const keys = ctx.result.keys
 const savingAccounts = sInput.map((_, i) => keys[i])
 const sLength = sInput.length
 const checkingAccounts = cInput.map((_, i) => keys[sLength + i])
 return { savingAccounts, checkingAccounts }
}
```

リゾルバーを保存し、 AWS AppSync コンソールのクエリセクションに移動して、アカウントを入力します。

次のミューテーションを実行します。

```
mutation populateAccounts {
  populateAccounts (
    savingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 100},
      {accountNumber: "2", username: "Amy", balance: 90},
      {accountNumber: "3", username: "Lily", balance: 80},
    ]
    checkingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 70},
      {accountNumber: "2", username: "Amy", balance: 60},
      {accountNumber: "3", username: "Lily", balance: 50},
    ]) {
    savingAccounts {
      accountNumber
    }
    checkingAccounts {
      accountNumber
    }
  }
}
```

1 つのミューテーションで 3 つの普通預金口座と 3 つの当座預金口座に入力しました。

DynamoDB コンソールを使用して、savingAccounts テーブルと checkingAccounts テーブルの両方にデータが表示されることを確認します。

TransactWriteItems - 振替

次のコードを使用して transferMoney ミューテーションにリゾルバーをアタッチします。送金ごとに、当座預金口座と普通預金口座の両方に成功修飾子が必要で、取引中の送金を追跡する必要があります。

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const transactions = ctx.args.transactions

 const savings = []
 const checkings = []
 const history = []
 transactions.forEach((t) => {
  const { savingAccountNumber, checkingAccountNumber, amount } = t
  savings.push({
   table: 'savingAccounts',
   operation: 'UpdateItem',
   key: util.dynamodb.toMapValues({ accountNumber: savingAccountNumber }),
   update: {
    expression: 'SET balance = balance - :amount',
    expressionValues: util.dynamodb.toMapValues({ ':amount': amount }),
   },
  })
  checkings.push({
   table: 'checkingAccounts',
   operation: 'UpdateItem',
   key: util.dynamodb.toMapValues({ accountNumber: checkingAccountNumber }),
   update: {
    expression: 'SET balance = balance + :amount',
    expressionValues: util.dynamodb.toMapValues({ ':amount': amount }),
   },
  })
  history.push({
   table: 'transactionHistory',
   operation: 'PutItem',
   key: util.dynamodb.toMapValues({ transactionId: util.autoId() }),
   attributeValues: util.dynamodb.toMapValues({
    from: savingAccountNumber,
    to: checkingAccountNumber,
    amount,
   }),
```

```
  })
 })

 return {
  version: '2018-05-29',
  operation: 'TransactWriteItems',
  transactItems: [...savings, ...checkings, ...history],
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
 }
 const tInput = ctx.args.transactions
 const tLength = tInput.length
 const keys = ctx.result.keys
 const savingAccounts = tInput.map((_, i) => keys[tLength * 0 + i])
 const checkingAccounts = tInput.map((_, i) => keys[tLength * 1 + i])
 const transactionHistory = tInput.map((_, i) => keys[tLength * 2 + i])
 return { savingAccounts, checkingAccounts, transactionHistory }
}
```

次に、 AWS AppSync コンソールのクエリセクションに移動し、次のように transferMoney ミューテーションを実行します。

```
mutation write {
  transferMoney(
    transactions: [
      {savingAccountNumber: "1", checkingAccountNumber: "1", amount: 7.5},
      {savingAccountNumber: "2", checkingAccountNumber: "2", amount: 6.0},
      {savingAccountNumber: "3", checkingAccountNumber: "3", amount: 3.3}
    ]) {
    savingAccounts {
      accountNumber
    }
    checkingAccounts {
      accountNumber
    }
    transactionHistory {
      transactionId
    }
  }
```

```
}
```

1 つのミューテーションで 2 つの銀行取引を送信しました。DynamoDB コンソールを使用して、savingAccounts、checkingAccounts、および transactionHistory の各テーブルにデータが表示されることを検証します。

TransactGetItems - 口座の取得

1 つの取引リクエストで普通預金口座と当座預金口座から詳細を取得するために、スキーマで `Query.getAccounts` GraphQL オペレーションにリゾルバーをアタッチします。[アタッチ] を選択し、次の画面で、このチュートリアルの最初で作成したのと同じ `TransactTutorial` データソースを選択します。以下のコードを使用します。

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { savingAccountNumbers, checkingAccountNumbers } = ctx.args

 const savings = savingAccountNumbers.map((accountNumber) => {
  return { table: 'savingAccounts', key: util.dynamodb.toMapValues({ accountNumber }) }
 })
 const checkings = checkingAccountNumbers.map((accountNumber) => {
  return { table: 'checkingAccounts', key:
 util.dynamodb.toMapValues({ accountNumber }) }
 })
 return {
  version: '2018-05-29',
  operation: 'TransactGetItems',
  transactItems: [...savings, ...checkings],
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
 }

 const { savingAccountNumbers: sInput, checkingAccountNumbers: cInput } = ctx.args
 const items = ctx.result.items
 const savingAccounts = sInput.map((_, i) => items[i])
 const sLength = sInput.length
 const checkingAccounts = cInput.map((_, i) => items[sLength + i])
```

```
 return { savingAccounts, checkingAccounts }
}
```

リゾルバーを保存し、AppSyncコンソールの[クエリ] AWS セクションに移動します。普通預金口座と当座預金口座を取得するには、次のクエリを実行します。

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
      username
      balance
    }
  }
}
```

以上で、 AWS AppSync を使用した DynamoDB のトランザクションのデモンストレーションが完了しました。

AWS AppSync での DynamoDB バッチオペレーションの使用

AWS AppSync は、1 つのリージョン内の 1 つ以上のテーブルで Amazon DynamoDB バッチオペレーションの使用をサポートしています。サポートされている処理は、`BatchGetItem`、`BatchPutItem`、および `BatchDeleteItem` です。 AWS AppSync でこれらの機能を使用すると、次のようなタスクを実行できます。

- 単一クエリでキーのリストを渡し、テーブルからの結果を返す
- 単一クエリで 1 つ以上のテーブルからレコードを読み取る
- 1 つ以上のテーブルに一連のレコードを書き込む
- 関連のある複数のテーブルのレコードを状況に応じて書き込む、または削除する

AWS AppSync のバッチオペレーションには、バッチ処理されていないオペレーションと 2 つの主な違いがあります。

- データソースのロールには、リゾルバーがアクセスするすべてのテーブルについてアクセス許可が必要です。
- リゾルバーのテーブル仕様はリクエストオブジェクトに含まれます。

1 つのテーブルのバッチ処理

> **Warning**
>
> `BatchPutItem` および `BatchDeleteItem` は、競合の検出と解決とともに使用する場合はサポートされていません。これらの設定は、起こり得るエラーを防ぐために無効にする必要があります。

はじめに、新しい GraphQL API を作成しましょう。 AWS AppSync コンソールで、API、GraphQL APIs、および設計を最初から作成を選択します。API に名前を付け`BatchTutorial API`、「次へ」を選択し、「GraphQL リソースを指定」ステップで「GraphQL リソースを後で作成」を選択して「次へ」をクリックします。詳細を確認して API を作成します。スキーマページに移動して以下のスキーマを張り付けます。クエリには ID のリストを渡すことに注意してください：

```
type Post {
    id: ID!
    title: String
}

input PostInput {
    id: ID!
    title: String
}

type Query {
    batchGet(ids: [ID]): [Post]
}

type Mutation {
    batchAdd(posts: [PostInput]): [Post]
    batchDelete(ids: [ID]): [Post]
```

```
}
```

スキーマを保存した後、ページ上部の [リソースを作成] を選択します。[既存のタイプを使用] を選択し、Post タイプを選択します。Posts テーブルに名前を付けます。プライマリキーが id に設定されていることを確認し、「GraphQL を自動生成」を選択解除し (独自のコードを指定します)、「作成」を選択します。はじめに、 AWS AppSync で新しい DynamoDB テーブルを作成し、そのテーブルに適切なロールで接続されたデータソースを作成します。ただし、ロールに追加する必要のあるアクセス許可がまだいくつかあります。[データソース] ページに移動し、新しいデータソースを選択します。[既存のロールを選択] で、テーブルのロールが自動的に作成されたことがわかります。ロール (appsync-ds-ddb-aaabbbcccddd-Posts のような表示になっているはずです) をメモし、IAM コンソール (https://console.aws.amazon.com/iam/) に移動します。IAM コンソールで [ロール] を選択し、テーブルからロールを選択します。自身のロールの [アクセス許可ポリシー] で、ポリシーの横にある [+] をクリックします (ロール名と似た名前になっているはずです)。ポリシーが表示されたら、折りたたみ可能なウィンドウの上部にある [編集] を選択します。ポリシーにバッチ許可、具体的にはdynamodb:BatchGetItem と dynamodb:BatchWriteItem を追加する必要があります。次のように表示されます。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:BatchGetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:…",
                "arn:aws:dynamodb:…"
            ]
        }
    ]
```

```
}
```

[次へ] を選択し、[変更を保存] を選択します。これで、ポリシーでバッチ処理が許可されるはずです。

AWS AppSync コンソールに戻り、スキーマページに移動し、Mutation.batchAddフィールドの横にあるアタッチを選択します。Posts テーブルをデータソースとして使用してリゾルバーを作成します。コードエディターで、ハンドラーを以下のスニペットに置き換えます。これは自動的に GraphQL の input PostInput 型に各項目を渡し、マップを作成します。このマップは BatchPutItem オペレーションに必要になります。

```
import { util } from "@aws-appsync/utils";

export function request(ctx) {
  return {
    operation: "BatchPutItem",
    tables: {
      Posts: ctx.args.posts.map((post) => util.dynamodb.toMapValues(post)),
    },
  };
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
}
```

AWS AppSync コンソールのクエリページに移動し、次のbatchAddミューテーションを実行します。

```
mutation add {
    batchAdd(posts:[{
            id: 1 title: "Running in the Park"},{
            id: 2 title: "Playing fetch"
        }]){
            id
            title
    }
```

```
}
```

結果が画面に表示されるはずです。これは、DynamoDB コンソールを確認して、Posts テーブルに書き込まれた値をスキャンすることで検証できます。

次に、Posts テーブルをデータソースとして使用する Query.batchGet フィールドに対してリゾルバーをアタッチするプロセスを繰り返します。ハンドラーを以下のコードに置き換えます。これは自動的に GraphQL の ids:[] 型に各項目を渡し、マップを作成します。このマップは BatchGetItem の処理に必要になります。

```
import { util } from "@aws-appsync/utils";

export function request(ctx) {
  return {
    operation: "BatchGetItem",
    tables: {
      Posts: {
        keys: ctx.args.ids.map((id) => util.dynamodb.toMapValues({ id })),
        consistentRead: true,
      },
    },
  };
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
}
```

次に、 AWS AppSync コンソールのクエリページに戻り、次のbatchGetクエリを実行します。

```
query get {
    batchGet(ids:[1,2,3]){
        id
        title
    }
}
```

これは、以前に追加した 2 つの id 値の結果を返します。値が null の id に対して 3 値が返っていることに注意してください。これは、その値に対応するレコードが Posts テーブルにまだないため

です。また、 AWS AppSync はクエリに渡されたキーと同じ順序で結果を返します。これは、 AWS AppSync がユーザーに代わって実行する追加機能です。したがって、batchGet(ids:[1,3,2]) に変更すると、順番が代わります。どの id で null 値が返されたのかもこれで分かります。

最後に、Posts テーブルをデータソースとして使用して、Mutation.batchDelete フィールドにもう 1 つリゾルバーをアタッチします。ハンドラーを以下のコードに置き換えます。これは自動的に GraphQL の ids:[] 型に各項目を渡し、マップを作成します。このマップは BatchGetItem の処理に必要になります。

```
import { util } from "@aws-appsync/utils";

export function request(ctx) {
  return {
    operation: "BatchDeleteItem",
    tables: {
      Posts: ctx.args.ids.map((id) => util.dynamodb.toMapValues({ id })),
    },
  };
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
}
```

次に、 AWS AppSync コンソールのクエリページに戻り、次のbatchDeleteミューテーションを実行します。

```
mutation delete {
    batchDelete(ids:[1,2]){ id }
}
```

id が 1 と 2 のレコードが削除されます。以前に使用した batchGet() クエリを再実行すると、これらは null を返します。

複数テーブルのバッチ処理

> **Warning**
>
> BatchPutItem および BatchDeleteItem は、競合の検出と解決とともに使用する場合はサポートされていません。これらの設定は、起こり得るエラーを防ぐために無効にする必要があります。

AWS AppSync では、テーブル間でバッチオペレーションを実行することもできます。より複雑なアプリケーションを作成しましょう。Pet Health アプリを作成するとします。これは、センサーによりペットの場所と体温を報告します。センサーは電池により動作し、数分ごとにネットワークへの接続を試みます。センサーが接続を確立すると、その測定値が AWS AppSync API に送信されます。その後、データの分析がトリガーされ、ペットの飼い主にダッシュボードが表示されます。センサーとバックエンドのデータストア間のやり取りの作成について考えてみましょう。

AWS AppSync コンソールで、API、GraphQL APIs、および設計を最初から作成を選択します。API に名前を付けMultiBatchTutorial API、「次へ」を選択し、「GraphQL リソースを指定」ステップで「GraphQL リソースを後で作成」を選択して「次へ」をクリックします。詳細を確認して API を作成します。「スキーマ」ページに移動し、次のスキーマを貼り付けて保存します。

```
type Mutation {
    # Register a batch of readings
    recordReadings(tempReadings: [TemperatureReadingInput], locReadings:
 [LocationReadingInput]): RecordResult
    # Delete a batch of readings
    deleteReadings(tempReadings: [TemperatureReadingInput], locReadings:
 [LocationReadingInput]): RecordResult
}

type Query {
    # Retrieve all possible readings recorded by a sensor at a specific time
    getReadings(sensorId: ID!, timestamp: String!): [SensorReading]
}

type RecordResult {
    temperatureReadings: [TemperatureReading]
    locationReadings: [LocationReading]
}

interface SensorReading {
```

```
    sensorId: ID!
    timestamp: String!
}

# Sensor reading representing the sensor temperature (in Fahrenheit)
type TemperatureReading implements SensorReading {
    sensorId: ID!
    timestamp: String!
    value: Float
}

# Sensor reading representing the sensor location (lat,long)
type LocationReading implements SensorReading {
    sensorId: ID!
    timestamp: String!
    lat: Float
    long: Float
}

input TemperatureReadingInput {
    sensorId: ID!
    timestamp: String
    value: Float
}

input LocationReadingInput {
    sensorId: ID!
    timestamp: String
    lat: Float
    long: Float
}
```

次の 2 つの DynamoDB テーブルを作成する必要があります。

- locationReadings はセンサー位置の読み取り値を格納します。
- temperatureReadings はセンサーの温度測定値を保存します。

両方のテーブルで同じプライマリキー構造体を共有します。sensorId（String）はパーティションキーで、timestamp（String）がソートキーです。

ページの上部で、[リソースの作成] を選択します。[既存のタイプを使用] を選択し、locationReadings タイプを選択します。locationReadings テーブルに名前を付けてくだ

さい。プライマリキー が sensorId に、ソートキーが timestamp に設定されていることを確認します。[GraphQL を自動的に生成する] (独自のコードを指定します) を選択解除し、[作成] を選択します。temperatureReadings をタイプとテーブル名として使用し,temperatureReadings に対してこのプロセスを繰り返して、ます。上記と同じキーを使用してください。

新しいテーブルには、自動的に生成されたロールが含まれます。これらのロールには、まだ追加する必要のあるアクセス許可がいくつかあります。データソースページに移動し、locationReadings を選択します。[既存のロールを選択] に、そのロールが表示されます。ロール (appsync-ds-ddb-aaabbbcccddd-locationReadings のような表示になっているはずです) をメモし、IAM コンソール (https://console.aws.amazon.com/iam/) に移動します。IAM コンソールで [ロール] を選択し、テーブルからロールを選択します。自身のロールの [アクセス許可ポリシー] で、ポリシーの横にある [+] をクリックします (ロール名と似た名前になっているはずです)。ポリシーが表示されたら、折りたたみ可能なウィンドウの上部にある [編集] を選択します。このポリシーにアクセス許可を追加する必要があります。次のように表示されます。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem",
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:account:table/locationReadings",
                "arn:aws:dynamodb:region:account:table/locationReadings/*",
                "arn:aws:dynamodb:us-east-1:account:table/temperatureReadings",
                "arn:aws:dynamodb:region:account:table/temperatureReadings/*"
            ]
        }
    ]
```

```
}
```

[次へ] を選択し、[変更を保存] を選択します。上記と同じポリシースニペットを使用して、temperatureReadings データソースに対してこのプロセスを繰り返します。

BatchPutItem - センサーの読み取り値の記録

センサーは、インターネットに接続後、読み取り値を送信できる必要があります。GraphQL の Mutation.recordReadings フィールドは、このために使用される API です。このフィールドにリゾルバーを追加する必要があります。

AWS AppSync コンソールのスキーマページで、 Mutation.recordReadingsフィールドの横にあるアタッチを選択します。次の画面で、locationReadings テーブルをデータソースとして使用してリゾルバーを作成します。

リゾルバーを作成したら、エディタでハンドラーを次のコードに置き換えます。このBatchPutItem操作により、複数のテーブルを指定できます。

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { locReadings, tempReadings } = ctx.args
 const locationReadings = locReadings.map((loc) => util.dynamodb.toMapValues(loc))
 const temperatureReadings = tempReadings.map((tmp) => util.dynamodb.toMapValues(tmp))

 return {
  operation: 'BatchPutItem',
  tables: {
   locationReadings,
   temperatureReadings,
  },
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.appendError(ctx.error.message, ctx.error.type)
 }
 return ctx.result.data
}
```

バッチ処理では、呼び出しから エラーと結果の両方が返る可能性があります。その場合は、任意に追加のエラー処理を実行できます。

> **Note**
>
> `utils.appendError()` の使用法は `util.error()` と同様ですが、リクエストまたはレスポンスハンドラーの評価を中断しないという大きな違いがあります。代わりに、エラーが発生したことをフィールドで通知します。ただし、テンプレートを評価して、データを引き続き呼び出し元に返すことも可能です。アプリケーションで部分的な結果を返す必要がある場合は、`utils.appendError()` を使用することを推奨します。

リゾルバーを保存し、 AWS AppSync コンソールのクエリページに移動します。これで、センサーの読み取り値をいくつか送信できます。

次のミューテーションを実行します。

```
mutation sendReadings {
  recordReadings(
    tempReadings: [
      {sensorId: 1, value: 85.5, timestamp: "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, value: 85.7, timestamp: "2018-02-01T17:21:06.000+08:00"},
      {sensorId: 1, value: 85.8, timestamp: "2018-02-01T17:21:07.000+08:00"},
      {sensorId: 1, value: 84.2, timestamp: "2018-02-01T17:21:08.000+08:00"},
      {sensorId: 1, value: 81.5, timestamp: "2018-02-01T17:21:09.000+08:00"}
    ]
    locReadings: [
      {sensorId: 1, lat: 47.615063, long: -122.333551, timestamp:
 "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, lat: 47.615163, long: -122.333552, timestamp:
 "2018-02-01T17:21:06.000+08:00"},
      {sensorId: 1, lat: 47.615263, long: -122.333553, timestamp:
 "2018-02-01T17:21:07.000+08:00"},
      {sensorId: 1, lat: 47.615363, long: -122.333554, timestamp:
 "2018-02-01T17:21:08.000+08:00"},
      {sensorId: 1, lat: 47.615463, long: -122.333555, timestamp:
 "2018-02-01T17:21:09.000+08:00"}
    ]) {
    locationReadings {
      sensorId
      timestamp
      lat
```

```
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}
```

1 つのミューテーションで 10 個のセンサー読み取り値を送信しました。これらは 2 つのテーブルに分けられています。DynamoDB コンソールを使用して、locationReadings テーブルと temperatureReadings テーブルの両方にデータが表示されることを確認します。

BatchDeleteItem - センサーの読み取り値の削除

同様に、センサーの読み取り値を一括して削除する必要もあります。これを行うために、Mutation.deleteReadings GraphQL フィールドを使用してみましょう。 AWS AppSync コンソールのスキーマページで、 Mutation.deleteReadingsフィールドの横にあるアタッチを選択します。次の画面で、locationReadings テーブルをデータソースとして使用してリゾルバーを作成します。

リゾルバーを作成したら、コードエディターのハンドラーを以下のスニペットに置き換えます。このリゾルバーでは、提供された入力から sensorId と timestamp を抽出する補助関数マッパーを使用します。

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { locReadings, tempReadings } = ctx.args
 const mapper = ({ sensorId, timestamp }) => util.dynamodb.toMapValues({ sensorId,
 timestamp })

 return {
  operation: 'BatchDeleteItem',
  tables: {
   locationReadings: locReadings.map(mapper),
   temperatureReadings: tempReadings.map(mapper),
  },
 }
}
```

```
export function response(ctx) {
 if (ctx.error) {
  util.appendError(ctx.error.message, ctx.error.type)
 }
 return ctx.result.data
}
```

リゾルバーを保存し、 AWS AppSync コンソールのクエリページに移動します。では、いくつかのセンサーの読み取り値を削除しましょう!

次のミューテーションを実行します。

```
mutation deleteReadings {
  # Let's delete the first two readings we recorded
  deleteReadings(
    tempReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]
    locReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}
```

> **Note**
>
> `DeleteItem` オペレーションとは対照的に、完全に削除された項目はレスポンスで返されません。渡されたキーのみが返されます。詳細については、DynamoDB の JavaScript リゾルバー関数リファレンスの BatchDeleteItem を参照してください。

DynamoDB コンソールを通じて `locationReadings` および `temperatureReadings` テーブルから削除されたこれら 2 つの読み取り値を検証します。

BatchGetItem - 読み取り値の取得

アプリのもう 1 つの一般的な処理として、特定の時刻にセンサーの読み取り値を取得します。スキーマの Query.getReadings GraphQL フィールドにリゾルバーをアタッチしましょう。 AWS AppSync コンソールのスキーマページで、 Query.getReadingsフィールドの横にあるアタッチを選択します。次の画面で、locationReadings テーブルをデータソースとして使用してリゾルバーを作成します。

以下のコードを使用します。

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const keys = [util.dynamodb.toMapValues(ctx.args)]
 const consistentRead = true
 return {
  operation: 'BatchGetItem',
  tables: {
   locationReadings: { keys, consistentRead },
   temperatureReadings: { keys, consistentRead },
  },
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.appendError(ctx.error.message, ctx.error.type)
 }
 const { locationReadings: locs, temperatureReadings: temps } = ctx.result.data

 return [
  ...locs.map((l) => ({ ...l, __typename: 'LocationReading' })),
  ...temps.map((t) => ({ ...t, __typename: 'TemperatureReading' })),
 ]
}
```

リゾルバーを保存し、 AWS AppSync コンソールのクエリページに移動します。では、センサーの読み取り値を取得しましょう!

次のクエリを実行します。

```
query getReadingsForSensorAndTime {
```

```
  # Let's retrieve the very first two readings
  getReadings(sensorId: 1, timestamp: "2018-02-01T17:21:06.000+08:00") {
    sensorId
    timestamp
    ...on TemperatureReading {
      value
    }
    ...on LocationReading {
      lat
      long
    }
  }
}
```

以上で、 AWS AppSync を使用した DynamoDB のバッチ処理のデモンストレーションが完了しました。

エラー処理

In AWS AppSync では、データソースオペレーションが部分的な結果を返すことがあります。部分的な結果という用語は、処理の出力がいくつかのデータと 1 つのエラーで構成されていることを表す場合に使用します。エラー処理は本質的にアプリケーション固有であるため、 AWS AppSync はレスポンスハンドラーでエラーを処理する機会を提供します。リゾルバーの呼び出しエラーがある場合、これは context から `ctx.error` として読み出せます。呼び出しエラーには必ずメッセージと型が含まれています。これらは `ctx.error.message` と `ctx.error.type` というプロパティとしてアクセスできます。レスポンスハンドラーでは、部分的な結果を次の 3 つの方法で処理できます。

1. データを返すだけで、呼び出しエラーは通知しない
2. ハンドラーの評価を停止することでエラーを発生させる (`util.error(...)` を使用)。データは返さない。
3. エラーを付加し (`util.appendError(...)` を使用)、データも返す

DynamoDB のバッチ処理を使用して、上記の 3 つの方法をそれぞれ試してみましょう!

DynamoDB のバッチ処理

DynamoDB のバッチ処理を使用すると、バッチを部分的に完了させることができます。つまり、リクエストされた項目またはキーの一部を未処理のままにすることができます。 AWS AppSync がバッチを完了できない場合、未処理の項目と呼び出しエラーがコンテキストに設定されます。

このチュートリアルの以前のセクションの Query.getReadings 処理で使用した BatchGetItem フィールドの設定を使用してエラー処理を実装します。ここでは、Query.getReadings フィールドの実行中に、temperatureReadings DynamoDB テーブルがプロビジョニングされたスループットを使い果たしたとします。DynamoDB は、 AWS AppSync による 2 回目の試行ProvisionedThroughputExceededException中に を生成し、バッチ内の残りの要素を処理しました。

次の JSON は、DynamoDB のバッチ処理の呼び出し後、レスポンスハンドラーの呼び出し前にシリアル化された context を表しています。

```
{
  "arguments": {
    "sensorId": "1",
    "timestamp": "2018-02-01T17:21:05.000+08:00"
  },
  "source": null,
  "result": {
    "data": {
      "temperatureReadings": [
        null
      ],
      "locationReadings": [
        {
          "lat": 47.615063,
          "long": -122.333551,
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ]
    },
    "unprocessedKeys": {
      "temperatureReadings": [
        {
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ],
      "locationReadings": []
    }
  },
  "error": {
    "type": "DynamoDB:ProvisionedThroughputExceededException",
```

```
    "message": "You exceeded your maximum allowed provisioned throughput for a table or
 for one or more global secondary indexes. (...)"
  },
  "outErrors": []
}
```

context に関する注意事項がいくつかあります。

- 呼び出しエラーは ctx.error by AWS AppSync のコンテキストで設定され、エラータイプは に設定されていますDynamoDB:ProvisionedThroughputExceededException。
- エラーが存在していても、結果はテーブルごとに ctx.result.data にマッピングされます。
- 未処理のキーは ctx.result.data.unprocessedKeys でアクセス可能です。ここでは、テーブルスループットが不十分なため、 AWS AppSync はキー (sensorId:1、timestamp:2018-02-01T17:21:05.000+08:00) を持つ項目を取得できませんでした。

> **Note**
>
> BatchPutItem の場合、これは ctx.result.data.unprocessedItems です。BatchDeleteItem の場合、これは ctx.result.data.unprocessedKeys です。

3 つの異なる方法でこのエラーを処理しましょう。

1. 呼び出しエラーを通知しない

呼び出しエラーを処理せずにデータを返して、エラーを実質的に通知しません。指定した GraphQL フィールドの結果は常に成功とします。

記述するコードは通常どおりであり、結果のデータのみにフォーカスします。

レスポンスハンドラー

```
export function response(ctx) {
  return ctx.result.data
}
```

GraphQL レスポンス

```
{
  "data": {
```

```
    "getReadings": [
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "lat": 47.615063,
        "long": -122.333551
      },
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  }
}
```

データのみが有効なため、エラーレスポンスにエラーは追加されません。

2. エラーを発生させて、レスポンスハンドラーの実行を中止する

クライアントから見て、部分的な障害を完全な障害として扱う必要がある場合、レスポンスハンドラーの実行を中断することでデータの返送を抑制することができます。この動作には、`util.error(...)` ユーティリティメソッドを使用するのが最適です。

レスポンスハンドラーコード

```
export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type, null,
 ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}
```

GraphQL レスポンス

```
{
  "data": {
    "getReadings": null
  },
  "errors": [
    {
      "path": [
```

```
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
        }
      ],
      "message": "You exceeded your maximum allowed provisioned throughput for a table
 or for one or more global secondary indexes. (...)"
    }
  ]
}
```

一部の結果が DynamoDB のバッチ処理から返されてもエラーが発生するように選択して、getReadings GraphQL フィールドが null になり、かつ GraphQL レスポンスの errors ブロックにエラーが追加されるようにします。

3. エラーを付加してデータとエラーの両方を返す

場合によっては、より優れたユーザーエクスペリエンスを提供するために、アプリケーションから部分的な結果を返すとともに、クライアントに未処理の項目を通知することができます。クライアントでは、再試行を実装することも、エラーを変換してエンドユーザーに通知することもできます。util.appendError(...) はこの動作を可能とするユーティリティメソッドであり、アプリケーションの設計者は、レスポンスハンドラーの評価を妨げることなく、context にエラーを付加できます。レスポンスハンドラーを評価した後、 AWS AppSync はコンテキストエラーを GraphQL レスポンスのエラーブロックに追加して処理します。

レスポンスハンドラーコード

```
export function response(ctx) {
```

```
  if (ctx.error) {
    util.appendError(ctx.error.message, ctx.error.type, null,
 ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}
```

ここでは、GraphQL レスポンスのエラーブロック内の呼び出しエラーと unprocessedKeys 要素の両方が転送されています。以下のレスポンスで分かるとおり、getReadings フィールドもまた locationReadings テーブルから部分的なデータを返します。

GraphQL レスポンス

```
{
  "data": {
    "getReadings": [
      null,
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
```

```
          "column": 3
        }
      ],
      "message": "You exceeded your maximum allowed provisioned throughput for a table
 or for one or more global secondary indexes. (...)"
    }
  ]
}
```

AWS AppSync での HTTP リゾルバーの使用

AWS AppSync を使用すると、サポートされているデータソース (Amazon DynamoDB AWS Lambda、Amazon OpenSearch Service、または Amazon Aurora) を使用して、GraphQL フィールドを解決するための任意の HTTP エンドポイントに加えて、さまざまなオペレーションを実行できます。HTTP エンドポイントが利用可能になったら、データソースを使用してこれに接続できます。その後、クエリ、ミューテーション、およびサブスクリプションなどの GraphQL オペレーションを実行するために、スキーマ内のリゾルバーを設定できます。このチュートリアルでは、いくつかの一般的な例を説明します。

このチュートリアルでは、 AWS AppSync GraphQL エンドポイントで REST API (Amazon API Gateway と Lambda により作成) を使用します。

REST API を作成する

次の AWS CloudFormation テンプレートを使用して、このチュートリアルで動作する REST エンドポイントを設定できます。

AWS CloudFormation スタックは次のステップを実行します。

1. マイクロサービス用のビジネスロジックを含む Lambda 関数を設定します。
2. 以下のエンドポイント/メソッド/コンテンツタイプを組み合わせて、API Gateway REST API をセットアップします。

API リソースパス	HTTP メソッド	サポートされているコンテンツタイプ
/v1/users	POST	application/json
/v1/users	GET	application/json
/v1/users/1	GET	application/json
/v1/users/1	PUT	application/json
/v1/users/1	DELETE	application/json

GraphQL API の作成

AWS AppSync で GraphQL API を作成するには：

1. AWS AppSync コンソールを開き、API の作成を選択します。
2. [GraphQL API] を選択し、[最初から設計] を選択します。[次へ] を選択します。
3. API 名として「`UserData`」と入力します。[次へ] を選択します。
4. `Create GraphQL resources later` を選択してください。[次へ] を選択します。
5. 入力内容を確認し、[API の作成] を選択します。

AWS AppSync コンソールは、API キー認証モードを使用して新しい GraphQL API を作成します。コンソールを使用して GraphQL API をさらに設定し、リクエストを実行できます。

GraphQL スキーマを作成する

GraphQL API を作成できたので、次に GraphQL スキーマを作成します。 AWS AppSync コンソールのスキーマエディタで、以下のスニペットを使用します。

```
type Mutation {
    addUser(userInput: UserInput!): User
    deleteUser(id: ID!): User
}

type Query {
```

```
    getUser(id: ID): User
    listUser: [User!]!
}

type User {
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
}

input UserInput {
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
}
```

HTTP データソースを設定する

HTTP データソースを設定するには、以下の操作を行います。

1. AWS AppSync GraphQL API のデータソースページで、データソースの作成を選択します。
2. HTTP_Example のようにデータソースの名前を入力します。
3. [データソースタイプ] で [HTTP エンドポイント] を選択します。
4. チュートリアルの最初に作成された API ゲートウェイエンドポイントに、エンドポイントを設定します。Lambda コンソールに移動し、[アプリケーション] でアプリケーションを見つけると、スタックで生成されたエンドポイントを見つけることができます。アプリケーションの設定の中に、エンドポイントとなる API エンドポイントが表示されます。エンドポイントの一部にステージ名が含まれていないことを確認します。例えば、エンドポイントが `https://aaabbbcccd.execute-api.us-east-1.amazonaws.com/v1` であれば、`https://aaabbbcccd.execute-api.us-east-1.amazonaws.com` に入力します。

> **Note**
>
> 現時点では、 AWS AppSync ではパブリックエンドポイントのみがサポートされています。

AWS AppSync サービスによって認識される認証機関の詳細については、「HTTPS エンドポイント AWS AppSync の によって認識される認証機関 (CA)」を参照してください。

リゾルバーの設定

このステップでは、http データソースを とgetUser および addUser クエリに接続します。

getUserリゾルバーをセットアップするには

1. AWS AppSync GraphQL API で、スキーマタブを選択します。
2. スキーマエディタの右側の [リゾルバー] ペインの [クエリタイプ] で、getUserフィールドを見つけて [アタッチ] を選択します。
3. リゾルバータイプはUnitのままにし、ランタイムはAPPSYNC_JSにします。
4. [データソース名] で、先ほど作成した HTTP エンドポイントを選択します。
5. [作成] を選択します。
6. Resolver コードエディターで、次のスニペットをリクエストハンドラーとして追加します。

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 return {
  version: '2018-05-29',
  method: 'GET',
  params: {
   headers: {
    'Content-Type': 'application/json',
   },
  },
  resourcePath: `/v1/users/${ctx.args.id}`,
 }
}
```

7. 次のスニペットをレスポンスハンドラーとして追加します。

```
export function response(ctx) {
 const { statusCode, body } = ctx.result
 // if response is 200, return the response
 if (statusCode === 200) {
```

```
  return JSON.parse(body)
 }
 // if response is not 200, append the response to error block.
 util.appendError(body, statusCode)
}
```

8. [Query (クエリ)] タブを選択して、以下のクエリを実行します。

```
query GetUser{
    getUser(id:1){
        id
        username
    }
}
```

これは以下のレスポンスを返します。

```
{
    "data": {
        "getUser": {
            "id": "1",
            "username": "nadia"
        }
    }
}
```

リゾルバーをセットアップするには、以下の手順に従います。

1. [Schema (スキーマ)] タブを選択します。
2. スキーマエディタの右側の [リゾルバー] ペインの [クエリタイプ] で、addUserフィールドを見つけて [アタッチ] を選択します。
3. リゾルバータイプはUnitのままにし、ランタイムはAPPSYNC_JSにします。
4. [データソース名] で、先ほど作成した HTTP エンドポイントを選択します。
5. [作成] を選択します。
6. Resolver コードエディターで、次のスニペットをリクエストハンドラーとして追加します。

```
export function request(ctx) {
    return {
        "version": "2018-05-29",
```

```
        "method": "POST",
        "resourcePath": "/v1/users",
        "params":{
            "headers":{
                "Content-Type": "application/json"
            },
        "body": ctx.args.userInput
        }
    }
}
```

7. 次のスニペットをレスポンスハンドラーとして追加します。

```
export function response(ctx) {
    if(ctx.error) {
        return util.error(ctx.error.message, ctx.error.type)
    }
    if (ctx.result.statusCode == 200) {
        return ctx.result.body
    } else {
        return util.appendError(ctx.result.body, "ctx.result.statusCode")
    }
}
```

8. [Query (クエリ)] タブを選択して、以下のクエリを実行します。

```
mutation addUser{
    addUser(userInput:{
        id:"2",
        username:"shaggy"
    }){
        id
        username
    }
}
```

getUser クエリを再実行すると、次のレスポンスが返されるはずです。

```
{
    "data": {
        "getUser": {
        "id": "2",
        "username": "shaggy"
```

```
        }
    }
}
```

AWS サービスの呼び出し

HTTP リゾルバーを使用して、 AWS サービスの GraphQL API インターフェイスを設定できます。への HTTP リクエストは、 AWS が送信者を識別できるように、署名バージョン 4 プロセスで署名 AWS する必要があります。 AWS AppSync は、IAM ロールを HTTP データソースに関連付けるときに、ユーザーに代わって署名を計算します。

HTTP リゾルバーを使用して AWS サービスを呼び出すには、次の 2 つの追加コンポーネントを提供します。

- AWS サービス APIs を呼び出すアクセス許可を持つ IAM ロール
- データソースの署名設定

たとえば、HTTP リゾルバーを使用して ListGraphqlApis オペレーションを呼び出す場合は、まず AWS AppSync が引き受ける IAM ロールを作成し、次のポリシーをアタッチします。

JSON

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

次に、 AWS AppSync の HTTP データソースを作成します。この例では、米国西部 (オレゴン) リージョンで AWS AppSync を呼び出します。http.json という名前のファイルで、署名リージョンとサービス名を含む以下の HTTP 設定を定義します。

```
{
    "endpoint": "https://appsync.us-west-2.amazonaws.com/",
    "authorizationConfig": {
        "authorizationType": "AWS_IAM",
        "awsIamConfig": {
            "signingRegion": "us-west-2",
            "signingServiceName": "appsync"
        }
    }
}
```

次に、 を使用して AWS CLI 、次のように関連付けられたロールを持つデータソースを作成します。

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

リゾルバーをスキーマの フィールドにアタッチする場合は、次のリクエストマッピングテンプレートを使用して AWS AppSync を呼び出します。

```
{
    "version": "2018-05-29",
    "method": "GET",
    "resourcePath": "/v1/apis"
}
```

このデータソースに対して GraphQL クエリを実行すると、 AWS AppSync は指定したロールを使用してリクエストに署名し、リクエストに署名を含めます。クエリは、その AWS リージョンのアカウントの AWS AppSync GraphQL APIsのリストを返します。

AWS AppSyncの Data API での Aurora PostgreSQL の使用

を使用して GraphQL API を Aurora PostgreSQL データベースに接続する方法について説明します AWS AppSync。この統合により、GraphQL オペレーションを通じて SQL クエリとミューテーションを実行して、スケーラブルなデータ駆動型アプリケーションを構築できます。 は、Data API で有効になっている Amazon Aurora クラスターに対して SQL ステートメントを実行するためのデータソース AWS AppSync を提供します。 AWS AppSync リゾルバーを使用して、GraphQL クエリ、

ミューテーション、サブスクリプションを含むデータ API に対して SQL ステートメントを実行できます。

このチュートリアルを開始する前に、 AWS サービスと GraphQL の概念に関する基本的な知識が必要です。

> **Note**
>
> このチュートリアルでは、US-EAST-1 リージョンを使用しています。

トピック

- Aurora PostgreSQL データベースをセットアップする
- データベースとテーブルの作成
- GraphQL スキーマを作成する
- RDS のリゾルバー
- クラスターの削除

Aurora PostgreSQL データベースをセットアップする

Amazon RDS データソースを に追加する前に AWS AppSync、以下を実行します。

1. Aurora Serverless v2 クラスターで Data API を有効にします。
2. を使用してシークレットを設定する AWS Secrets Manager
3. 次の AWS CLI コマンドを使用してクラスターを作成します。

```
aws rds create-db-cluster \
            --db-cluster-identifier appsync-tutorial \
            --engine aurora-postgresql \
            --engine-version 16.6 \
            --serverless-v2-scaling-configuration MinCapacity=0,MaxCapacity=1 \
            --master-username USERNAME \
            --master-user-password COMPLEX_PASSWORD \
            --enable-http-endpoint
```

これにより、クラスターの ARN が返されます。クラスターを作成したら、次の AWS CLI コマンドを使用して Serverless v2 インスタンスを追加する必要があります。

```
aws rds create-db-instance \
    --db-cluster-identifier appsync-tutorial \
    --db-instance-identifier appsync-tutorial-instance-1 \
    --db-instance-class db.serverless \
    --engine aurora-postgresql
```

> **Note**
>
> これらのエンドポイントがアクティブ化されるまでに時間がかかります。ステータスは、クラスターの接続とセキュリティタブの RDS コンソールで確認できます。

次の AWS CLI コマンドを使用して、クラスターのステータスを確認します。

```
aws rds describe-db-clusters \
    --db-cluster-identifier appsync-tutorial \
    --query "DBClusters[0].Status"
```

AWS Secrets Manager コンソールまたは を使用して、前のステップCOMPLEX_PASSWORDの USERNAMEと を使用して、次のような入力ファイル AWS CLI を使用してシークレットを作成します。

```
{
    "username": "USERNAME",
    "password": "COMPLEX_PASSWORD"
}
```

これをパラメータとして に渡します AWS CLI。

```
aws secretsmanager create-secret \
    --name appsync-tutorial-rds-secret \
    --secret-string file://creds.json
```

これにより、シークレットの ARN が返されます。 AWS AppSync コンソールでデータソースを作成するときは、Aurora Serverless v2 クラスターの ARN と のシークレットを書き留めます。

データベースとテーブルの作成

まず、 という名前のデータベースを作成しますTESTDB。PostgreSQL では、データベースはテーブルやその他の SQL オブジェクトを保持するコンテナです。API に追加 AWS AppSync する前

に、Aurora Serverless v2 クラスターが正しく設定されていることを確認します。まず、次のように--sqlパラメータを使用して TESTDB データベースを作成します。

```
aws rds-data execute-statement \
    --resource-arn "arn:aws:rds:us-east-1:111122223333 ISN:cluster:appsync-tutorial" \
    --secret-arn "arn:aws:secretsmanager:us-east-1:111122223333 ISN:secret:appsync-
tutorial-rds-secret" \
    --sql "create DATABASE \"testdb\"" \
    --database "postgres"
```

これがエラーなしで実行されたら、create table コマンドを使用して 2 つのテーブルを追加します。

```
 aws rds-data execute-statement \
    --resource-arn "arn:aws:rds:us-east-1:111122223333 ISN:cluster:appsync-tutorial" \
    --secret-arn "arn:aws:secretsmanager:us-east-1:111122223333 ISN:secret:appsync-
tutorial-rds-secret" \
    --database "testdb" \
    --sql 'create table public.todos (id serial constraint todos_pk primary key,
 description text not null, due date not null, "createdAt" timestamp default now());'

aws rds-data execute-statement \
    --resource-arn "arn:aws:rds:us-east-1:111122223333 ISN:cluster:appsync-tutorial" \
    --secret-arn "arn:aws:secretsmanager:us-east-1:111122223333 ISN:secret:appsync-
tutorial-rds-secret" \
    --database "testdb" \
    --sql 'create table public.tasks (id serial constraint tasks_pk primary key,
 description varchar, "todoId" integer not null constraint tasks_todos_id_fk references
 public.todos);'
```

成功したら、クラスターを API のデータソースとして追加します。

GraphQL スキーマを作成する

Aurora Serverless v2 Data API が設定されたテーブルで実行されたので、GraphQL スキーマを作成します。API 作成ウィザードを使用して既存のデータベースからテーブル設定をインポートすることで、API をすばやく作成できます。

開始方法

1. AWS AppSync コンソールで API の作成を選択し、Amazon Aurora クラスターから開始します。

2. [API 名] などの API の詳細を指定し、API を生成するデータベースを選択します。
3. データベースを選択します。必要に応じてリージョンを更新し、Aurora クラスターと TESTDB データベースを選択します。
4. シークレットを選択し、[インポート] を選択します。
5. テーブルが検出されたら、型名を更新します。Todos を Todo に変更し、Tasks を Task に変更します。
6. [スキーマをプレビュー] を選択して、生成されたスキーマをプレビューします。スキーマは以下のようになります。

```
type Todo {
  id: Int!
  description: String!
  due: AWSDate!
  createdAt: String
}

type Task {
  id: Int!
  todoId: Int!
  description: String
}
```

7. ロールの場合、 で AWS AppSync 新しいロールを作成するか、次のようなポリシーで作成できます。

   JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "rds-data:ExecuteStatement"
            ],
            "Resource": [
                "arn:aws:rds:us-east-1:111122223333 ISN:cluster:appsync-
tutorial",
                "arn:aws:rds:us-east-1:111122223333 ISN:cluster:appsync-
tutorial:*"
            ]
```

```
        },
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": [
                "arn:aws:secretsmanager:us-east-1:111122223333
 ISN:secret:your:secret:arn:appsync-tutorial-rds-secret",
                "arn:aws:secretsmanager:us-east-1:111122223333
 ISN:secret:your:secret:arn:appsync-tutorial-rds-secret:*"
            ]
        }
    ]
}
```

このポリシーには、ロールにアクセス許可を付与するステートメントが 2 つあることに注意してください。最初のリソースは Aurora クラスターで、2 番目のリソースは AWS Secrets Manager ARN です。

[次へ] を選択し、設定の詳細を確認して [API の作成] を選択します。これで完全に動作する API が作成されました。API の全詳細は、[スキーマ] ページで確認できます。

RDS のリゾルバー

API 作成フローでは、型とやり取りするリゾルバーが自動的に作成されました。スキーマページを見ると、リゾルバーの一部が表示されます。

- `Mutation.createTodo` フィールドで `todo` を作成する。
- `Mutation.updateTodo` フィールドで `todo` を更新する。
- `Mutation.deleteTodo` フィールドで `todo` を削除する。
- `Query.getTodo` フィールドで 単一の `todo` を取得する。
- `Query.listTodos` フィールドで `todos` をすべて一覧表示する。

`Task` 型にアタッチされた類似のフィールドとリゾルバーがあります。いくつかのリゾルバーを詳しく見ていきましょう。

Mutation.createTodo

AWS AppSync コンソールのスキーマエディタの右側で、 testdbの横にある を選択しますcreateTodo(...): Todo。リゾルバーコードは、rds モジュールの insert 関数を使用して、todos テーブルにデータを追加する挿入ステートメントを動的に作成します。ここではPostgres を使用しているため、挿入されたデータをreturning ステートメントを活用して返すことができます。

次のリゾルバーを更新して、 dueフィールドDATEのタイプを適切に指定します。

```
import { util } from '@aws-appsync/utils';
import { insert, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/
rds';

export function request(ctx) {
    const { input } = ctx.args;
    // if a due date is provided, cast is as `DATE`
    if (input.due) {
        input.due = typeHint.DATE(input.due)
    }
    const insertStatement = insert({
        table: 'todos',
        values: input,
        returning: '*',
    });
    return createPgStatement(insertStatement)
}

export function response(ctx) {
    const { error, result } = ctx;
    if (error) {
        return util.appendError(
            error.message,
            error.type,
            result
        )
    }
    return toJsonObject(result)[0][0]
}
```

リゾルバーを保存します。型ヒントは、入力オブジェクト内の due をDATE 型として適切にマークします。これにより、Postgres エンジンが値を適切に解釈できるようになります。次に、スキーマ

を更新して CreateTodo 入力から id を削除します。Postgres データベースは生成された ID を返すことができるため、次のように 1 回のリクエストで作成して返すことができます。

```
input CreateTodoInput {
    due: AWSDate!
    createdAt: String
    description: String!
}
```

変更を加え、スキーマを更新します。クエリエディタに移動して、次のようにデータベースに項目を追加します。

```
mutation CreateTodo {
  createTodo(input: {description: "Hello World!", due: "2023-12-31"}) {
    id
    due
    description
    createdAt
  }
}
```

次の結果が得られます。

```
{
  "data": {
    "createTodo": {
      "id": 1,
      "due": "2023-12-31",
      "description": "Hello World!",
      "createdAt": "2023-11-14 20:47:11.875428"
    }
  }
}
```

Query.listTodos

コンソールのスキーマエディタの右側で、listTodos(id: ID!): Todo の横にある testdb を選択します。リクエストハンドラーは select ユーティリティ関数を使用して、実行時にリクエストを動的に構築します。

```
export function request(ctx) {
```

```
    const { filter = {}, limit = 100, nextToken } = ctx.args;
    const offset = nextToken ? +util.base64Decode(nextToken) : 0;
    const statement = select({
        table: 'todos',
        columns: '*',
        limit,
        offset,
        where: filter,
    });
    return createPgStatement(statement)
}
```

due の日付に基づいて todos をフィルタリングしたいとします。due の値を DATE にキャストするようにリゾルバーを更新しましょう。インポートのリストとリクエストハンドラーを次のように更新します。

```
import { util } from '@aws-appsync/utils';
import * as rds from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { filter: where = {}, limit = 100, nextToken } = ctx.args;
  const offset = nextToken ? +util.base64Decode(nextToken) : 0;

  // if `due` is used in a filter, CAST the values to DATE.
  if (where.due) {
    Object.entries(where.due).forEach(([k, v]) => {
      if (k === 'between') {
        where.due[k] = v.map((d) => rds.typeHint.DATE(d));
      } else {
        where.due[k] = rds.typeHint.DATE(v);
      }
    });
  }

  const statement = rds.select({
    table: 'todos',
    columns: '*',
    limit,
    offset,
    where,
  });
  return rds.createPgStatement(statement);
}
```

```
export function response(ctx) {
  const {
    args: { limit = 100, nextToken },
    error,
    result,
  } = ctx;
  if (error) {
    return util.appendError(error.message, error.type, result);
  }
  const offset = nextToken ? +util.base64Decode(nextToken) : 0;
  const items = rds.toJsonObject(result)[0];
  const endOfResults = items?.length < limit;
  const token = endOfResults ? null : util.base64Encode(`${offset + limit}`);
  return { items, nextToken: token };
}
```

クエリエディタで、次の操作を行います。

```
query LIST {
  listTodos(limit: 10, filter: {due: {between: ["2021-01-01", "2025-01-02"]}}) {
    items {
      id
      due
      description
    }
  }
}
```

Mutation.updateTodo

Todo を update することも可能です。[クエリ] エディタから、id 1 の最初の Todo 項目を更新しましょう。

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits"}) {
    description
    due
    id
  }
}
```

更新する項目の id を指定する必要があることに注意してください。条件を指定して、特定の条件を満たす項目のみを更新することもできます。たとえば、説明がedits次のように で始まる場合にのみ項目を編集できます。

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits: make a change"}, condition:
 {description: {beginsWith: "edits"}}) {
    description
    due
    id
  }
}
```

create オペレーションと list オペレーションを処理したのと同じように、リゾルバーを更新して due フィールドを DATE にキャストできます。これらの変更をupdateTodo次のように に保存します。

```
import { util } from '@aws-appsync/utils';
import * as rds from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: { id, ...values }, condition = {}, } = ctx.args;
  const where = { ...condition, id: { eq: id } };

  // if `due` is used in a condition, CAST the values to DATE.
  if (condition.due) {
    Object.entries(condition.due).forEach(([k, v]) => {
      if (k === 'between') {
        condition.due[k] = v.map((d) => rds.typeHint.DATE(d));
      } else {
        condition.due[k] = rds.typeHint.DATE(v);
      }
    });
  }

  // if a due date is provided, cast is as `DATE`
  if (values.due) {
    values.due = rds.typeHint.DATE(values.due);
  }

  const updateStatement = rds.update({
    table: 'todos',
```

```
    values,
    where,
    returning: '*',
  });
  return rds.createPgStatement(updateStatement);
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    return util.appendError(error.message, error.type, result);
  }
  return rds.toJsonObject(result)[0][0];
}
```

次に、以下の条件で更新を試してください。

```
mutation UPDATE {
  updateTodo(
    input: {
        id: 1, description: "edits: make a change", due: "2023-12-12"},
    condition: {
        description: {beginsWith: "edits"}, due: {ge: "2023-11-08"}})
    {
          description
          due
          id
        }
}
```

Mutation.deleteTodo

deleteTodo ミューテーションで Todo を delete できます。これはミューupdateTodoテーションと同様に動作し、削除する項目の id を次のように指定する必要があります。

```
mutation DELETE {
  deleteTodo(input: {id: 1}) {
    description
    due
    id
  }
}
```

カスタムクエリの記述

rds モジュールユーティリティを使用して SQL ステートメントを作成しました。独自でカスタムした静的ステートメントを記述して、データベースとやり取りすることもできます。まず、スキーマを更新して CreateTask 入力から id フィールドを削除します。

```
input CreateTaskInput {
    todoId: Int!
    description: String
}
```

次に、いくつかのタスクを作成します。タスクには、Todo次のように との外部キー関係があります。

```
mutation TASKS {
  a: createTask(input: {todoId: 2, description: "my first sub task"}) { id }
  b:createTask(input: {todoId: 2, description: "another sub task"}) { id }
  c: createTask(input: {todoId: 2, description: "a final sub task"}) { id }
}
```

getTodoAndTasks次のように、 Queryというタイプに新しいフィールドを作成します。

```
getTodoAndTasks(id: Int!): Todo
```

次のように Todoタイプに tasks フィールドを追加します。

```
type Todo {
    due: AWSDate!
    id: Int!
    createdAt: String
    description: String!
    tasks:TaskConnection
}
```

スキーマを保存します。コンソールのスキーマエディタの右側で、getTodosAndTasks(id: Int!): Todo に対して [リゾルバーをアタッチ] を選択します。Amazon RDS データソースを選択します。次のコードでリゾルバーを更新します。

```
import { sql, createPgStatement,toJsonObject } from '@aws-appsync/utils/rds';
```

```
export function request(ctx) {
    return createPgStatement(
        sql`SELECT * from todos where id = ${ctx.args.id}`,
        sql`SELECT * from tasks where "todoId" = ${ctx.args.id}`);
}

export function response(ctx) {
    const result = toJsonObject(ctx.result);
    const todo = result[0][0];
    if (!todo) {
        return null;
    }
    todo.tasks = { items: result[1] };
    return todo;
}
```

このコードでは、sql タグテンプレートを使用して、実行時に動的な値を安全に渡すことができる SQL ステートメントを記述します。createPgStatement は、一度に最大 2 つの SQL リクエストを受け入れることができます。これを利用して、todo に対して 1 つのクエリを送信し、tasks に対して別のクエリを送信します。これは、JOIN ステートメントやその他の方法を使用して行うこともできます。つまり、独自の SQL ステートメントを記述してビジネスロジックを実装できるということです。クエリエディタでクエリを使用するには、次の手順を実行します。

```
query TodoAndTasks {
  getTodosAndTasks(id: 2) {
    id
    due
    description
    tasks {
      items {
        id
        description
      }
    }
  }
}
```

クラスターの削除

> **Important**
>
> クラスターは完全に削除されます。このアクションを実行する前に、プロジェクトを徹底的に確認してください。

クラスターを削除するには

```
$ aws rds delete-db-cluster \
    --db-cluster-identifier appsync-tutorial \
    --skip-final-snapshot
```

AWS AppSyncの VTL リゾルバーチュートリアル

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

データソースとリゾルバーは AWS AppSync によって GraphQL リクエストを変換し、 AWS リソースから情報を取得するために使用されます。 AWS AppSync は、特定のデータソースタイプとの自動プロビジョニングと接続をサポートしています。 AWS AppSync は AWS Lambda、データソースとして Amazon DynamoDB、リレーショナルデータベース (Amazon Aurora Serverless)、Amazon OpenSearch Service、HTTP エンドポイントもサポートしています。既存の AWS リソースで GraphQL API を使用するか、データソースとリゾルバーをゼロから構築できます。以下のセクションは、より一般的な GraphQL のユースケースをチュートリアルの形式で説明することを目的としています。

AWS AppSync は、リゾルバーの Apache Velocity Template Language (VTL) で記述されたマッピングテンプレートを使用します。 マッピングテンプレートの詳細については、「リゾルバーのマッピングテンプレートリファレンス」を参照してください。VTL の操作の詳細については、「リゾルバマッピングテンプレートプログラミングガイド」を参照してください。

AWS AppSync は、「スキーマからプロビジョニングする (オプション) 」および「サンプルスキーマを起動する」で説明されているように、GraphQL スキーマからの DynamoDB テーブルの自動プロビジョニングをサポートしています。既存の DynamoDB テーブルからインポートして、スキーマを作成し、リゾルバーを接続することもできます。それについては、「(オプション) Amazon DynamoDB からのインポート」で説明しています。

トピック

- DynamoDB リゾルバーを使用したシンプルなポストアプリケーションの作成
- での AWS Lambda リゾルバーの使用 AWS AppSync
- AWS AppSync での Amazon OpenSearch Service リゾルバーの使用
- AWS AppSync でのローカルリゾルバーの使用
- AWS AppSyncでの GraphQL リゾルバーの組み合わせ
- AWS AppSync での DynamoDB バッチオペレーションの使用

- AWS AppSync での DynamoDB トランザクションの実行
- AWS AppSync での HTTP リゾルバーの使用
- での Aurora Serverless v2 の使用 AWS AppSync
- AWS AppSync でのパイプラインリゾルバーの使用
- のバージョニングされたデータソースでの Delta Sync オペレーションの使用 AWS AppSync

DynamoDB リゾルバーを使用したシンプルなポストアプリケーションの作成

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

このチュートリアルでは、独自の Amazon DynamoDB テーブルを AWS AppSync に持ち込み、GraphQL API に接続する方法を示します。

ユーザーに代わって DynamoDB リソースをプロビジョニングすることを AWS AppSync に許可できます。または、必要であれば、データソースとリゾルバーを作成することで、既存のテーブルを GraphQL スキーマに接続できます。いずれの場合も、GraphQL ステートメントを使用して DynamoDB データベースへの読み取りと書き込みを行うことができ、リアルタイムデータをサブスクライブできます。

GraphQL ステートメントが DynamoDB オペレーションに変換され、レスポンスが GraphQL に変換されるように、特定の設定ステップを完了しておく必要があります。このチュートリアルでは、いくつかの実際のシナリオおよびデータアクセスパターンを使用して、その設定手順の概要を示します。

DynamoDB テーブルのセットアップ

このチュートリアルを開始するには、まず以下の手順に従って AWS リソースをプロビジョニングする必要があります。

1. CLI で次の AWS CloudFormation テンプレートを使用して AWS リソースをプロビジョニングします。

```
aws cloudformation create-stack \
```

```
    --stack-name AWSAppSyncTutorialForAmazonDynamoDB \
    --template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/
dynamodb/AmazonDynamoDBCFTemplate.yaml \
    --capabilities CAPABILITY_NAMED_IAM
```

または、アカウントの米国西部 2 (オレゴン) リージョンで次の AWS CloudFormation スタックを起動することもできます AWS 。

これにより、以下の項目が作成されます。

- データを保持するDynamoDBテーブルがAppSyncTutorial-Post呼び出されます。Post
- AWS AppSync がPostテーブルとやり取りできるようにする IAM ロールおよび関連する IAM 管理ポリシー。

2. スタックおよび作成されたリソースに関する詳細を確認するには、次の CLI コマンドを実行します。

```
aws cloudformation describe-stacks --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

3. そのリソースを削除するには、次のコマンドを実行します。

```
aws cloudformation delete-stack --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

GraphQL API の作成

AWS AppSync で GraphQL API を作成するには：

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   - API ダッシュボードで、[API の作成] を選択します。
2. [API をカスタマイズまたは Amazon DynamoDB からインポート] ウィンドウで、[一から構築] を選択します。
   - 同じウィンドウの右にある [開始] を選択します。
3. [API 名] フィールドで、API の名前を AWSAppSyncTutorial に設定します。
4. [作成] を選択します。

AWS AppSync コンソールは、API キー認証モードを使用して新しい GraphQL API を作成します。このコンソールを使用して、残りの GraphQL API をセットアップでき、このチュートリアルの残りの部分でクエリを実行できます。

基本的な Post API の定義

AWS AppSync GraphQL API を作成したので、ポストデータの基本的な作成、取得、削除を許可する基本スキーマを設定できます。

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   - API ダッシュボードで、先ほど作成した API を選択します。
2. サイドバーで [スキーマ] を選択します。
   - [スキーマ] ペインで、内容を次のコードに置き換えます。

```
schema {
    query: Query
    mutation: Mutation
}

type Query {
    getPost(id: ID): Post
}

type Mutation {
    addPost(
        id: ID!
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}

type Post {
    id: ID!
    author: String
    title: String
    content: String
    url: String
    ups: Int!
```

```
    downs: Int!
    version: Int!
}
```

3. [保存] を選択します。

このスキーマでは、Post 型と、Post オブジェクトを追加および取得するオペレーションを定義しています。

DynamoDB テーブル用のデータソースの設定

次に、スキーマで定義されているクエリとミューテーションを AppSyncTutorial-Post DynamoDB テーブルにリンクします。

まず、 AWS AppSync はテーブルを認識する必要があります。これを行うには AWS AppSync にデータソースをセットアップします。

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   a. API ダッシュボードで、GraphQL API を選択します。
   b. サイドバー で [データソース] を選択します。
2. [データソースを作成] を選択します。
   a. データソース名には、PostDynamoDBTable を入力します。
   b. [データソースタイプ] として [Amazon DynamoDB テーブル] を選択します。
   c. [リージョン] として [US-WEST-2] を選択します。
   d. [テーブル名] には、[AppSyncTutorial-Post] DynamoDB テーブルを選択します。
   e. 次に、新しい IAM ロールを作成するか (推奨)、lambda:invokeFunction への IAM アクセス許可を持つ既存のロールを選択します。データソースのアタッチセクションで説明しているように、既存のロールには信頼ポリシーが必要です。

      次に、リソースで操作を実行するために必要なアクセス許可を持つ IAM ポリシーの例を示します。

      JSON

```
{
     "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [ "lambda:invokeFunction" ],
            "Resource": [
                "arn:aws:lambda:us-
west-2:123456789012:function:myFunction",
                "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:*"
            ]
        }
    ]
}
```

3. [作成] を選択します。

addPost リゾルバー (DynamoDB PutItem) のセットアップ

AWS AppSync が DynamoDB テーブルを認識したら、リゾルバーを定義して個々のクエリとミューテーションにリンクできます。最初に作成するリゾルバーは addPost リゾルバーです。このリゾルバーによって、ユーザーが AppSyncTutorial-Post DynamoDB テーブルにポストを作成できるようになります。

リゾルバーには以下のコンポーネントがあります。

- リゾルバーをアタッチする、GraphQL スキーマ内の場所。この例では、addPost 型の Mutation フィールドにリゾルバーをセットアップしています。このリゾルバーは、呼び出し元が `mutation { addPost(...){...} }` を呼び出したときに呼び出されます。
- このリゾルバーで使用するデータソース。この例では、PostDynamoDBTable DynamoDB テーブルにエントリを追加できるように、前に定義した AppSyncTutorial-Post データソースを使用します。
- リクエストマッピングテンプレート。リクエストマッピングテンプレートの目的は、発信者からの受信リクエストを受け取り、それを AWS AppSync が DynamoDB に対して実行する手順に変換することです。
- レスポンスマッピングテンプレート。レスポンスマッピングテンプレートの目的は、DynamoDB からのレスポンスを取り込み、それを GraphQL で想定されているものに変換し直すことです。これは、DynamoDB でのデータのシェイプが GraphQL での Post 型と異なる場合に便利です。ただし、この例では、両方のシェイプが同じであるため、データをそのまま渡します。

リゾルバーをセットアップするには、以下の手順に従います。

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   a. API ダッシュボードで、GraphQL API を選択します。
   b. サイドバー で [データソース] を選択します。
2. [データソースを作成] を選択します。
   a. データソース名には、`PostDynamoDBTable` を入力します。
   b. [データソースタイプ] として [Amazon DynamoDB テーブル] を選択します。
   c. [リージョン] として [US-WEST-2] を選択します。
   d. [テーブル名] には、[AppSyncTutorial-Post] DynamoDB テーブルを選択します。
   e. 次に、新しい IAM ロールを作成するか (推奨)、`lambda:invokeFunction` への IAM アクセス許可を持つ既存のロールを選択します。データソースのアタッチセクションで説明しているように、既存のロールには信頼ポリシーが必要です。

      次に、リソースで操作を実行するために必要なアクセス許可を持つ IAM ポリシーの例を示します。

      JSON

```
{
     "Version": "2012-10-17",
     "Statement": [
         {
             "Effect": "Allow",
             "Action": [ "lambda:invokeFunction" ],
             "Resource": [
                 "arn:aws:lambda:us-
west-2:123456789012:function:myFunction",
                 "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:*"
             ]
         }
     ]
 }
```

3. [作成] を選択します。
4. [Schema (スキーマ)] タブを選択します。

5. 右側のデータ型ペインで、ミューテーション型のaddPostフィールドを見つけて、アタッチを選択します。
6. [アクション] メニューで [ランタイムの更新] を選択し、[ユニットリゾルバー (VTL のみ)] を選択します。
7. [データソース名] で、[PostDynamoDBTable] を選択します。
8. 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "attributeValues" : {
        "author" : $util.dynamodb.toDynamoDBJson($context.arguments.author),
        "title" : $util.dynamodb.toDynamoDBJson($context.arguments.title),
        "content" : $util.dynamodb.toDynamoDBJson($context.arguments.content),
        "url" : $util.dynamodb.toDynamoDBJson($context.arguments.url),
        "ups" : { "N" : 1 },
        "downs" : { "N" : 0 },
        "version" : { "N" : 1 }
    }
}
```

注 :「型」はすべてのキーと属性値で指定されています。例えば、author フィールドを { "S" : "${context.arguments.author}" } に設定します。S 部分は、値が文字列値になることを to AWS AppSync と DynamoDB に示します。実際の値は author 引数から入力されます。同様に、version フィールドは、型として N が使用されているため数値フィールドです。最後に、ups、downs および version フィールドの初期化も行っています。

このチュートリアルでは、DynamoDB に挿入される新しい項目のインデックスを作成する GraphQL ID!タイプが、 という名前の自動 ID 生成用のユーティリティを持つ client arguments. AWS AppSync comes の一部として提供されることを指定しました。$utils.autoId()これは、 の形式でも使用できます"id" : { "S" : "${$utils.autoId()}" }。そのため、id: ID! を addPost() のスキーマ定義から除外するだけで、自動的に挿入されます。このチュートリアルではこの手法を使用しませんが、DynamoDB テーブルに書き込む場合はこの手法を検討することをお勧めします。

マッピングテンプレートの詳細については、「リゾルバーのマッピングテンプレートの概要」リファレンスドキュメントを参照してください。GetItem リクエストマッピングの詳細については、「GetItem」リファレンスドキュメントを参照してください。型の詳細については、「型システム (リクエストマッピング)」リファレンスドキュメントを参照してください。

9. 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
$utils.toJson($context.result)
```

注 : AppSyncTutorial-Post テーブルでのデータのシェイプは GraphQL での Post 型のシェイプと厳密に一致しているため、このレスポンスマッピングテンプレートは結果をそのまま渡すだけです。また、このチュートリアルのすべての例では、これと同じレスポンスマッピングテンプレートだけを使用しているため、作成するファイルはこの 1 つだけです。

10. [保存] を選択します。

ポストを追加する API の呼び出し

これでリゾルバーがセットアップされたので、 AWS AppSync は着信addPostミューテーションを DynamoDB PutItem オペレーションに変換できます。ユーザーはミューテーションを実行してテーブルに何かを入れることができるようになりました。

- [クエリ] タブを選択します。
- 以下のミューテーションを [クエリ] ペインに貼り付けます。

```
mutation addPost {
  addPost(
    id: 123
    author: "AUTHORNAME"
    title: "Our first post!"
    content: "This is our first post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
```

```
    version
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- 新しく作成されたポストの結果が、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

何が起こったのかを以下に説明します。

- AWS AppSync はaddPostミューテーションリクエストを受け取りました。
- AWS AppSync はリクエストとリクエストマッピングテンプレートを受け取り、リクエストマッピングドキュメントを生成しました。そのドキュメントは次のようになっていました。

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : { "S" : "123" }
    },
    "attributeValues" : {
        "author": { "S" : "AUTHORNAME" },
        "title": { "S" : "Our first post!" },
        "content": { "S" : "This is our first post." },
        "url": { "S" : "https://aws.amazon.com/appsync/" },
        "ups" : { "N" : 1 },
```

```
        "downs" : { "N" : 0 },
        "version" : { "N" : 1 }
    }
}
```

- AWS AppSync は、リクエストマッピングドキュメントを使用して DynamoDBPutItem リクエストを生成して実行しました。
- AWS AppSync はPutItemリクエストの結果を受け取り、GraphQL タイプに変換し直しました。

```
{
    "id" : "123",
    "author": "AUTHORNAME",
    "title": "Our first post!",
    "content": "This is our first post.",
    "url": "https://aws.amazon.com/appsync/",
    "ups" : 1,
    "downs" : 0,
    "version" : 1
}
```

- そのデータは、レスポンスマッピングドキュメントを介して変更されずに渡されました。
- 新しく作成されたオブジェクトが GraphQL レスポンスで返されました。

getPost リゾルバー (DynamoDB GetItem) のセットアップ

これで、AppSyncTutorial-Post DynamoDB テーブルにデータを追加できるようになりました。次は、AppSyncTutorial-Post テーブルからデータを取得できるように、getPost クエリを設定する必要があります。そのためには、別のリゾルバーを設定します。

- [Schema (スキーマ)] タブを選択します。
- 右側のデータ型ペインで、Query型のgetPostフィールドを見つけて、[アタッチ] を選択します。
- [アクション] メニューで [ランタイムの更新] を選択し、[ユニットリゾルバー (VTL のみ)] を選択します。
- [データソース名] で、[PostDynamoDBTable] を選択します。
- 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
```

```
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

- 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
$utils.toJson($context.result)
```

- [保存] を選択します。

ポストを取得する API の呼び出し

これでリゾルバーが設定され、 AWS AppSync は受信`getPost`クエリを DynamoDB`GetItem` オペレーションに変換する方法を知っています。次は、先ほど作成したポストを取得するクエリを実行します。

- [クエリ] タブを選択します。
- [Queries] ペインに、次の内容を貼り付けます。

```
query getPost {
  getPost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- DynamoDB から取得されたポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "getPost": {
      "id": "123",
```

```
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

何が起こったのかを以下に説明します。

- AWS AppSync はgetPostクエリリクエストを受信しました。
- AWS AppSync はリクエストとリクエストマッピングテンプレートを受け取り、リクエストマッピングドキュメントを生成しました。そのドキュメントは次のようになっていました。

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "123" }
    }
}
```

- AWS AppSync はリクエストマッピングドキュメントを使用して、DynamoDB GetItem リクエストを生成して実行しました。
- AWS AppSync はGetItemリクエストの結果を受け取り、GraphQL タイプに変換し直しました。

```
{
    "id" : "123",
    "author": "AUTHORNAME",
    "title": "Our first post!",
    "content": "This is our first post.",
    "url": "https://aws.amazon.com/appsync/",
    "ups" : 1,
    "downs" : 0,
    "version" : 1
}
```

- そのデータは、レスポンスマッピングドキュメントを介して変更されずに渡されました。

- 取得したオブジェクトがレスポンスで返されました。

別の方法として、次の例を指定します。

```
query getPost {
  getPost(id:123) {
    id
    author
    title
  }
}
```

getPost クエリに、id、author および title のみが必要な場合は、DynamoDB から AWS AppSyncへの不要なデータ転送を避けるため、投影式を使用して DynamoDB テーブルから必要な属性のみを指定するようにリクエストマッピングテンプレートを変更できます。例えば、リクエストマッピングテンプレートは以下のスニペットのようになります。

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "projection" : {
     "expression" : "#author, id, title",
     "expressionNames" : { "#author" : "author"}
    }
}
```

updatePost ミューテーション (DynamoDB UpdateItem) の作成

これで、DynamoDB 内の Post オブジェクトを作成および取得できるようになりました。次は、オブジェクトを更新できるように、新しいミューテーションを設定します。そのためには、DynamoDB の UpdateItem オペレーションを使用します。

- [Schema (スキーマ)] タブを選択します。
- [スキーマ] ペインの Mutation タイプを次のように変更して、新しい updatePost ミューテーションを追加します。

```
type Mutation {
```

```
    updatePost(
        id: ID!,
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post
    addPost(
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}
```

- [保存] を選択します。
- 右側のデータ型ペインで、updatePostフィールドに新しく作成されたミューテーション型を見つけて、アタッチを選択します。
- [アクション] メニューで [ランタイムの更新] を選択し、[ユニットリゾルバー (VTL のみ)] を選択します。
- [データソース名] で、[PostDynamoDBTable] を選択します。
- 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "SET author = :author, title = :title, content = :content,
 #url = :url ADD version :one",
        "expressionNames": {
            "#url" : "url"
        },
        "expressionValues": {
            ":author" : $util.dynamodb.toDynamoDBJson($context.arguments.author),
            ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title),
            ":content" : $util.dynamodb.toDynamoDBJson($context.arguments.content),
            ":url" : $util.dynamodb.toDynamoDBJson($context.arguments.url),
            ":one" : { "N": 1 }
```

```
        }
    }
}
```

注意: このリゾルバーでは DynamoDB の UpdateItem が使用されていて、そのオペレーションは PutItem オペレーションと大幅に異なります。DynamoDB では、項目全体が書き込まれるのではなく、特定の属性が更新されるようにします。これを行うには、DynamoDB の更新式を使用します。その式自体は、`expression` セクションの `update` フィールドで指定されています。その式では、`author`、`title`、`content`、url の各属性を設定し、`version` フィールドを増分しています。使用される値は式自体には記述されていません。この式には、名前がコロンで始まるプレースホルダーがあり、それぞれの値は `expressionValues` フィールドで定義されています。最後に、DynamoDB には、`expression` で使用できない予約語があります。例えば、`url` は予約語であるため、`url` フィールドを更新するには、名前のプレースホルダーを `expressionNames` フィールドで定義できます。

`UpdateItem` リクエストマッピングの詳細については、「UpdateItem」リファレンスドキュメントを参照してください。更新式の記述方法の詳細については、「DynamoDB UpdateExpressions」のドキュメントを参照してください。

- 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
$utils.toJson($context.result)
```

ポストを更新する API の呼び出し

これでリゾルバーが設定され、 AWS AppSync は着信`update`ミューテーションを DynamoDB`Update` オペレーションに変換する方法を知っています。ユーザーは、前のステップで書き込んだ項目を更新するミューテーションを実行できるようになりました。

- [クエリ] タブを選択します。
- 次のミューテーションを [Queries (クエリ)] ペインに貼り付けます。また、`id` 引数を、前にメモしておいた値に更新する必要があります。

```
mutation updatePost {
  updatePost(
    id:"123"
    author: "A new author"
    title: "An updated author!"
    content: "Now with updated content!"
```

```
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- DynamoDB で更新されたポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "updatePost": {
      "id": "123",
      "author": "A new author",
      "title": "An updated author!",
      "content": "Now with updated content!",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 2
    }
  }
}
```

この例では、リクエストマッピングテンプレートが AWS AppSync upsと DynamoDB にそれらのdownsフィールドの操作を要求しなかったため、 フィールドと フィールドは変更されませんでした。また、version フィールドが 1 ずつ増加します。これは、version フィールドに 1 を追加するように AWS AppSync と DynamoDB に指示しているためです。

updatePost リゾルバー (DynamoDB UpdateItem) の変更

updatePost ミューテーションの手始めとしてはこれで十分ですが、次の 2 つの主な問題があります。

- 1 つのフィールドだけを更新する場合でも、すべてのフィールドを更新する必要があります。
- 2 人のユーザーがこのオブジェクトを変更すると、情報が失われる可能性があります。

これらの問題に対処するために、リクエストで指定された引数のみを変更し、UpdateItem オペレーションに条件を追加するように、updatePost ミューテーションを修正します。

1. [Schema (スキーマ)] タブを選択します。
2. [Schema (スキーマ)] ペインで Mutation 型の updatePost フィールドを変更して、author、title、content、url の各引数から感嘆符を除去し、id フィールドが元のままになるようにします。これにより、それらの引数が省略可能になります。また、新しい必須の expectedVersion 引数を追加します。

```
type Mutation {
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}
```

3. [保存] を選択します。
4. 右側の [Data types (データ型)] ペインで、ミューテーション型の updatePost フィールドを見つけます。
5. PostDynamoDBTable を選択し、既存リゾルバーを開きます。
6. [Configure the request mapping template (リクエストマッピングテンプレートの設定)] にあるリクエストマッピングテンプレートを次のように変更します。

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
```

```
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },

    ## Set up some space to keep track of things you're updating **
    #set( $expNames  = {} )
    #set( $expValues = {} )
    #set( $expSet = {} )
    #set( $expAdd = {} )
    #set( $expRemove = [] )

    ## Increment "version" by 1 **
    $!{expAdd.put("version", ":one")}
    $!{expValues.put(":one", { "N" : 1 })}

    ## Iterate through each argument, skipping "id" and "expectedVersion" **
    #foreach( $entry in $context.arguments.entrySet() )
        #if( $entry.key != "id" && $entry.key != "expectedVersion" )
            #if( (!$entry.value) && ("$!{entry.value}" == "") )
                ## If the argument is set to "null", then remove that attribute from
the item in DynamoDB **

                #set( $discard = ${expRemove.add("#${entry.key}")} )
                $!{expNames.put("#${entry.key}", "$entry.key")}
            #else
                ## Otherwise set (or update) the attribute on the item in DynamoDB **

                $!{expSet.put("#${entry.key}", ":${entry.key}")}
                $!{expNames.put("#${entry.key}", "$entry.key")}
                $!{expValues.put(":${entry.key}", { "S" : "${entry.value}" })}
            #end
        #end
    #end

    ## Start building the update expression, starting with attributes you're going to
SET **
    #set( $expression = "" )
    #if( !${expSet.isEmpty()} )
        #set( $expression = "SET" )
        #foreach( $entry in $expSet.entrySet() )
            #set( $expression = "${expression} ${entry.key} = ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
```

```
        #end
    #end

    ## Continue building the update expression, adding attributes you're going to ADD
**
    #if( !${expAdd.isEmpty()} )
        #set( $expression = "${expression} ADD" )
        #foreach( $entry in $expAdd.entrySet() )
            #set( $expression = "${expression} ${entry.key} ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Continue building the update expression, adding attributes you're going to
REMOVE **
    #if( !${expRemove.isEmpty()} )
        #set( $expression = "${expression} REMOVE" )

        #foreach( $entry in $expRemove )
            #set( $expression = "${expression} ${entry}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Finally, write the update expression into the document, along with any
expressionNames and expressionValues **
    "update" : {
        "expression" : "${expression}"
        #if( !${expNames.isEmpty()} )
            ,"expressionNames" : $utils.toJson($expNames)
        #end
        #if( !${expValues.isEmpty()} )
            ,"expressionValues" : $utils.toJson($expValues)
        #end
    },

    "condition" : {
        "expression"       : "version = :expectedVersion",
        "expressionValues" : {
```

```
            ":expectedVersion" :
 $util.dynamodb.toDynamoDBJson($context.arguments.expectedVersion)
        }
    }
}
```

7. [保存] を選択します。

このテンプレートでは、より複雑な例を示します。マッピングテンプレートの力と柔軟性を示しています。すべての引数をループ処理し、id と expectedVersion をスキップしています。引数が何かに設定されている場合、DynamoDB のオブジェクトでその属性を更新するように AWS AppSync と DynamoDB に要求します。属性が null に設定されている場合、 AWS AppSync と DynamoDB にポストオブジェクトからその属性を削除するように要求します。引数が指定されていない場合、その属性は元のままになります。また、version フィールドが増分されます。

また、新しい condition セクションがあります。条件式を使用すると、オペレーションが実行される前に既に DynamoDB にあるオブジェクトの状態に基づいて、リクエストが成功するかどうかを AWS AppSync と DynamoDB に指示できます。この例では、DynamoDB に現在ある項目の version フィールドが expectedVersion 引数と厳密に一致する場合にのみ、UpdateItem リクエストが成功するように指示しています。

条件式の詳細については、「条件式」リファレンスドキュメントを参照してください。

ポストを更新する API の呼び出し

新しいリゾルバーで Post オブジェクトを更新してみましょう。

- [クエリ] タブを選択します。
- 以下のミューテーションを [クエリ] ペインに貼り付けます。また、id 引数を、前にメモしておいた値に更新する必要があります。

```
mutation updatePost {
  updatePost(
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 2
  ) {
    id
    author
    title
```

```
    content
    url
    ups
    downs
    version
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- DynamoDB で更新されたポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "updatePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 3
    }
  }
}
```

このリクエストでは、 AWS AppSync と DynamoDB に titleフィールドと contentフィールドのみを更新するように依頼しました。その他のフィールドは元のままです (増分する version フィールドは除く)。title 属性を新しい値に設定し、ポストから content 属性を削除しています。author、urlups、downs の各フィールドは変更されません。

リクエストはまったく同じままで、このミューテーションをもう一度実行してみます。次のようなレスポンスが表示されます。

```
{
  "data": {
    "updatePost": null
  },
  "errors": [
    {
```

```
      "path": [
        "updatePost"
      ],
      "data": {
        "id": "123",
        "author": "A new author",
        "title": "An empty story",
        "content": null,
        "url": "https://aws.amazon.com/appsync/",
        "ups": 1,
        "downs": 0,
        "version": 3
      },
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
    }
  ]
}
```

このリクエストは、条件式が false と評価されるため失敗します。

- このリクエストを最初に実行したときに、DynamoDB 内のこのポストの `version` フィールドの値は 2 であり、`expectedVersion` 引数と一致していました。このリクエストは成功し、DynamoDB で `version` フィールドが 3 に増分されました。
- このリクエストを 2 回目に実行したときに、DynamoDB 内のこのポストの `version` フィールドの値は 3 であり、`expectedVersion` 引数と一致していませんでした。

このパターンは通常、「楽観的ロック」と呼ばれます。

AWS AppSync DynamoDB リゾルバーの機能の 1 つは、DynamoDB のポストオブジェクトの現在の値を返すことです。そのことは、GraphQL レスポンスの `data` セクションの `errors` フィールドで確認できます。アプリケーションでは、この情報を使用して、どのように続行するかを決めることができます。この例では、DynamoDB 内のオブジェクトの `version` フィールドが 3 に設定されてい

ることを確認できるため、expectedVersion 引数を 3 に更新するだけで、このリクエストが再度成功するようになります。

条件チェックの失敗の処理の詳細については、マッピングテンプレートのリファレンスドキュメントの「条件式」を参照してください。

upvotePost と downvotePost ミューテーション (DynamoDB UpdateItem) の作成

Post 型には、賛成票と反対票を記録できる ups フィールドと downs フィールドがありますが、この API ではそのフィールドに対して何も行えません。ポストに賛成および反対するための、いくつかのミューテーションを追加してみましょう。

- [Schema (スキーマ)] タブを選択します。
- [スキーマ] ペインの Mutation タイプを変更して、新しい upvotePost と downvotePost ミューテーションを次のように追加します。

```
type Mutation {
    upvotePost(id: ID!): Post
    downvotePost(id: ID!): Post
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post!
}
```

- [保存] を選択します。
- 右側のデータ型ペインで、ミューテーション型の新しく作成された upvotePost フィールドを見つけて、[アタッチ] を選択します。

- [アクション] メニューで [ランタイムの更新] を選択し、[ユニットリゾルバー (VTL のみ)] を選択します。
- [データソース名] で、[PostDynamoDBTable] を選択します。
- 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "ADD ups :plusOne, version :plusOne",
        "expressionValues" : {
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
$utils.toJson($context.result)
```

- [保存] を選択します。
- 右側のデータ型ペインで、ミューテーション型の新しく作成された downvotePost フィールドを見つけて、[アタッチ] を選択します。
- [データソース名] で、[PostDynamoDBTable] を選択します。
- 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "ADD downs :plusOne, version :plusOne",
        "expressionValues" : {
            ":plusOne" : { "N" : 1 }
        }
```

```
    }
}
```

- 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
$utils.toJson($context.result)
```

- [保存] を選択します。

ポストに賛成および反対するための API の呼び出し

これで新しいリゾルバーが設定され、 AWS AppSync は受信upvotePostまたはdownvoteミューテーションを DynamoDB UpdateItem オペレーションに変換する方法を知っています。これで、先ほど作成したポストに賛成または反対するミューテーションを実行できるようになりました。

- [クエリ] タブを選択します。
- 以下のミューテーションを [クエリ] ペインに貼り付けます。また、id 引数を、前にメモしておいた値に更新する必要があります。

```
mutation votePost {
  upvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- そのポストが DynamoDB で更新され、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "upvotePost": {
      "id": "123",
      "author": "A new author",
```

```
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 0,
      "version": 4
    }
  }
}
```

- さらに数回、[Execute query (クエリを実行)] を選択します。このクエリを実行するたびに、ups フィールドと version フィールドが 1 つずつ増加することを確認できます。
- 次のようにクエリを変更し、downvotePost ミューテーションを呼び出します。

```
mutation votePost {
  downvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。今度は、このクエリを実行するたびに、downs フィールドと version フィールドが 1 つずつ増加することを確認できます。

```
{
  "data": {
    "downvotePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 4,
      "version": 12
```

```
    }
  }
}
```

deletePost リゾルバー (DynamoDB DeletePost) のセットアップ

次は、ポストを削除するミューテーションを設定します。そのためには、DynamoDB の `DeleteItem` オペレーションを使用します。

- [Schema (スキーマ)] タブを選択します。
- [スキーマ] ペインの `Mutation` タイプを次のように変更して、新しい `deletePost` ミューテーションを追加します。

```
type Mutation {
    deletePost(id: ID!, expectedVersion: Int): Post
    upvotePost(id: ID!): Post
    downvotePost(id: ID!): Post
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post!
}
```

今回は、`expectedVersion` フィールドを省略可能にしています。これについては、リクエストマッピングテンプレートを追加するときに説明します。

- [保存] を選択します。
- 右側のデータ型ペインで、ミューテーション型の新しく作成された delete フィールドを見つけて、[アタッチ] を選択します。

- [アクション] メニューで [ランタイムの更新] を選択し、[ユニットリゾルバー (VTL のみ)] を選択します。
- [データソース名] で、[PostDynamoDBTable] を選択します。
- 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
    "version" : "2017-02-28",
    "operation" : "DeleteItem",
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($context.arguments.id)
    }
    #if( $context.arguments.containsKey("expectedVersion") )
        ,"condition" : {
            "expression"       : "attribute_not_exists(id) OR version
 = :expectedVersion",
            "expressionValues" : {
                ":expectedVersion" :
 $util.dynamodb.toDynamoDBJson($context.arguments.expectedVersion)
            }
        }
    #end
}
```

注 : expectedVersion 引数は省略可能な引数です。呼び出し元がリクエストで expectedVersion 引数を設定していると、項目が既に削除されている場合、または DynamoDB 内のポストの version 属性が expectedVersion と完全に一致する場合にのみ、DeleteItem リクエストが成功することを許可する条件が、テンプレートによって追加されます。この引数が省略されている場合は、DeleteItem リクエストで条件式が指定されていません。version の値や項目が DynamoDB に存在するかどうかに関係なく、成功します。

- 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
$utils.toJson($context.result)
```

注意: 項目を削除する場合でも、その項目がまだ削除されていなければ、削除された項目を返すことができます。

- [保存] を選択します。

DeleteItem リクエストマッピングの詳細については、「DeleteItem」リファレンスドキュメントを参照してください。

ポストを削除する API の呼び出し

これでリゾルバーが設定され、 AWS AppSync は着信deleteミューテーションをDynamoDBDeleteItem オペレーションに変換する方法を知っています。ユーザーはミューテーションを実行してテーブル内の何かを削除できるようになりました。

- [クエリ] タブを選択します。
- 以下のミューテーションを [クエリ] ペインに貼り付けます。また、id 引数を、前にメモしておいた値に更新する必要があります。

```
mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- このポストが DynamoDB から削除されます。 AWS AppSync は、DynamoDB から削除された項目の値を返します。この値は、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 4,
```

```
      "version": 12
    }
  }
}
```

この値は、この deletePost 呼び出しによって実際に DynamoDB から項目が削除された場合にのみ返されます。

- [クエリを実行] を再度選択します。
- この呼び出しは成功しますが、値は返されません。

```
{
  "data": {
    "deletePost": null
  }
}
```

次は、expectedValue を指定して、ポストを削除してみましょう。ただし、これまで使用してきたポストは削除したため、まず新しいポストを作成する必要があります。

- 以下のミューテーションを [クエリ] ペインに貼り付けます。

```
mutation addPost {
  addPost(
    id:123
    author: "AUTHORNAME"
    title: "Our second post!"
    content: "A new post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
```

```
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- 新しく作成されたポストの結果が、クエリペインの右側にある結果ペインに表示されます。新しく作成されたオブジェクトの id を書き留めておきます。その値はすぐに必要になります。これは次のように表示されます。

```
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

次に、expectedVersion に間違った値を入れて、ポストを削除してみましょう。

- 以下のミューテーションを [クエリ] ペインに貼り付けます。また、id 引数を、前にメモしておいた値に更新する必要があります。

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
```

```
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。

```
{
  "data": {
    "deletePost": null
  },
  "errors": [
    {
      "path": [
        "deletePost"
      ],
      "data": {
        "id": "123",
        "author": "AUTHORNAME",
        "title": "Our second post!",
        "content": "A new post.",
        "url": "https://aws.amazon.com/appsync/",
        "ups": 1,
        "downs": 0,
        "version": 1
      },
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
    }
  ]
}
```

条件式が false と評価されるため、このリクエストは失敗しました。DynamoDB でのそのポストの `version` の値が、`expectedValue` 引数で指定したものと一致していないためです。そのオブジェクトの現在の値が、GraphQL レスポンスの `data` セクションの `errors` フィールドで返されます。

- `expectedVersion` を訂正して、このリクエストをもう一度試してみます。

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- 今回は、リクエストが成功し、DynamoDB から削除された値が返されています。

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

- [クエリを実行] を再度選択します。
- この呼び出しは成功しますが、そのポストが DynamoDB で既に削除されているため、今回は値が返されません。

```
{
  "data": {
```

```
    "deletePost": null
  }
}
```

allPost リゾルバー (DynamoDB Scan) のセットアップ

これまでのところ、APIは、見たい各投稿の id がわかっている場合にのみ便利です。テーブル内のすべてのポストを返す新しいリゾルバーを追加してみましょう。

- [Schema (スキーマ)] タブを選択します。
- [スキーマ] ペインの Query タイプを次のように変更して、新しい allPost クエリを追加します。

```
type Query {
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

- 新しい PaginationPosts 型を追加します。

```
type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}
```

- [保存] を選択します。
- 右側のデータ型ペインで、ミューテーション型の新しく作成された allPost フィールドを見つけて、[アタッチ] を選択します。
- [アクション] メニューで [ランタイムの更新] を選択し、[ユニットリゾルバー (VTL のみ)] を選択します。
- [データソース名] で、[PostDynamoDBTable] を選択します。
- 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
    #if( ${context.arguments.count} )
        ,"limit": $util.toJson($context.arguments.count)
    #end
```

```
    #if( ${context.arguments.nextToken} )
        ,"nextToken": $util.toJson($context.arguments.nextToken)
    #end
}
```

このリゾルバーには、省略可能な 2 つの引数があります。count では、1 回の呼び出しで返される項目の最大数を指定し、nextToken は、次の結果セットを取得するために使用できます (nextToken の値がどのように生成されるかについては、後で説明します)。

- 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

注 : このレスポンスマッピングテンプレートは、これまで他の例で使用したものとは異なります。allPost クエリの結果は PaginatedPosts であり、ポストのリストとページ分割トークンが含まれています。このオブジェクトの形状は、 AWS AppSync DynamoDB Resolver から返されるものと異なります。投稿のリストは AWS AppSync DynamoDB Resolver の結果itemsで呼び出されますが、 postsでは呼び出されますPaginatedPosts。

- [保存] を選択します。

Scan リクエストマッピングの詳細については、「Scan」リファレンスドキュメントを参照してください。

すべてのポストをスキャンする API の呼び出し

これでリゾルバーが設定され、 AWS AppSync は受信allPostクエリを DynamoDBScan オペレーションに変換する方法を知っています。ユーザーは、テーブルをスキャンしてすべてのポストを取得できるようになりました。

ただし、これまで使用してきたデータはすべて削除したため、これを試す前にテーブルにデータを入力しておく必要があります。

- [クエリ] タブを選択します。
- 以下のミューテーションを [クエリ] ペインに貼り付けます。

```
mutation addPost {
  post1: addPost(id:1 author: "AUTHORNAME" title: "A series of posts, Volume 1"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post2: addPost(id:2 author: "AUTHORNAME" title: "A series of posts, Volume 2"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post3: addPost(id:3 author: "AUTHORNAME" title: "A series of posts, Volume 3"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post4: addPost(id:4 author: "AUTHORNAME" title: "A series of posts, Volume 4"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post5: addPost(id:5 author: "AUTHORNAME" title: "A series of posts, Volume 5"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post6: addPost(id:6 author: "AUTHORNAME" title: "A series of posts, Volume 6"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post7: addPost(id:7 author: "AUTHORNAME" title: "A series of posts, Volume 7"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post8: addPost(id:8 author: "AUTHORNAME" title: "A series of posts, Volume 8"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post9: addPost(id:9 author: "AUTHORNAME" title: "A series of posts, Volume 9"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。

では、テーブルをスキャンして、一度に 5 個の結果を返しましょう。

- 以下のクエリを [クエリ] ペインに貼り付けます。

```
query allPost {
  allPost(count: 5) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- 最初の 5 個のポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        },
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        }
      ],
      "nextToken":
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
    }
  }
}
```

5 つの結果と nextToken を取得しました。このトークンを使用して、次の結果セットを取得できます。

- 前回の結果セットからの allPost を含めるように、nextToken クエリを更新します。

```
query allPost {
  allPost(
    count: 5
    nextToken:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
```

```
  ) {
    posts {
      id
      author
    }
    nextToken
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- 残りの 4 個のポストが、クエリペインの右側にある結果ペインに表示されます。9 個のポストのすべてをページ分割して、ポストは残っていないため、この結果セットに nextToken はありません。これは次のように表示されます。

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        }
      ],
      "nextToken": null
    }
  }
}
```

「allPostsByAuthor」リゾルバー (DynamoDB クエリ) のセットアップ

DynamoDB ですべてのポストをスキャンだけでなく、特定の作成者が作成したポストを取得するクエリを DynamoDB に対して実行することもできます。前の手順で作成した DynamoDB テーブルには、既に `author-index` という `GlobalSecondaryIndex` があるため、DynamoDB の `Query` オペレーションでそれを使用して、特定の作成者が作成したすべてのポストを取得できます

- [Schema (スキーマ)] タブを選択します。
- [スキーマ] ペインの `Query` タイプを次のように変更して、新しい `allPostsByAuthor` クエリを追加します。

```
type Query {
    allPostsByAuthor(author: String!, count: Int, nextToken: String): PaginatedPosts!
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

  注 : このクエリでは、`allPost` クエリで使用したのと同じ `PaginatedPosts` 型を使用しています。
- [保存] を選択します。
- 右側のデータ型ペインで、ミューテーション型の新しく作成された allPostsByAuthor フィールドを見つけて、[アタッチ] を選択します。
- [アクション] メニューで [ランタイムの更新] を選択し、[ユニットリゾルバー (VTL のみ)] を選択します。
- [データソース名] で、[PostDynamoDBTable] を選択します。
- 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "index" : "author-index",
    "query" : {
      "expression": "author = :author",
        "expressionValues" : {
          ":author" : $util.dynamodb.toDynamoDBJson($context.arguments.author)
        }
    }
    #if( ${context.arguments.count} )
```

```
        ,"limit": $util.toJson($context.arguments.count)
    #end
    #if( ${context.arguments.nextToken} )
        ,"nextToken": "${context.arguments.nextToken}"
    #end
}
```

allPost リゾルバーと同様に、このリゾルバーには、省略可能な 2 つの引数があります。count では、1 回の呼び出しで返される項目の最大数を指定し、nextToken は、次の結果セットを取得するために使用できます (nextToken の値は前回の呼び出しから取得できます)。

- 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

注 : これは、allPost リゾルバーで使用したのと同じレスポンスマッピングテンプレートです。

- [保存] を選択します。

Query リクエストマッピングの詳細については、「クエリ」リファレンスドキュメントを参照してください。

特定の作成者によるすべてのポストをクエリする API の呼び出し

これでリゾルバーが設定され、 AWS AppSync は受信allPostsByAuthorミューテーションをauthor-indexインデックスに対する DynamoDBQuery オペレーションに変換する方法を知っています。ユーザーは、テーブルをクエリして、特定の作成者によるポストをすべて取得できます。

ただし、これまで使用していたポストはすべて同じ作成者だったため、それを行う前に、テーブルにポストを追加しておきましょう。

- [クエリ] タブを選択します。
- 以下のミューテーションを [クエリ] ペインに貼り付けます。

```
mutation addPost {
```

```
  post1: addPost(id:10 author: "Nadia" title: "The cutest dog in the world" content:
 "So cute. So very, very cute." url: "https://aws.amazon.com/appsync/" ) { author,
 title }
  post2: addPost(id:11 author: "Nadia" title: "Did you know...?" content: "AppSync
 works offline?" url: "https://aws.amazon.com/appsync/" ) { author, title }
  post3: addPost(id:12 author: "Steve" title: "I like GraphQL" content: "It's great"
 url: "https://aws.amazon.com/appsync/" ) { author, title }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。

では、Nadia が作成したすべてのポストを返すクエリを実行しましょう。

- 以下のクエリを [クエリ] ペインに貼り付けます。

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- Nadia が作成したすべてのポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
```

```
      "nextToken": null
    }
  }
}
```

Query でのページ分割は Scan とまったく同じように動作します。例えば、AUTHORNAME によるすべてのポストを検索して、一度に 5 個ずつ取得します。

- 以下のクエリを [クエリ] ペインに貼り付けます。

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- AUTHORNAME が作成したすべてのポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "2",
```

```
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        }
      ],
      "nextToken":
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
    }
  }
}
```

- 次のように、nextToken 引数を、前回のクエリで返された値に更新します。

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
    nextToken:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- AUTHORNAME が作成した残りのポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
```

```
          "id": "8",
          "title": "A series of posts, Volume 8"
        },
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        }
      ],
      "nextToken": null
    }
  }
}
```

セット型の使用

ここまでの Post 型は、フラットなキーと値のオブジェクトでした。セット、リスト、マップなど、AWS AppSyncDynamoDB リゾルバーを使用して複雑なオブジェクトをモデル化することもできます。

Post 型を更新して、タグを含めましょう。1 つのポストには、DynamoDB に文字列として保存されているタグを 0 個以上付けることができます。タグを追加および削除するミューテーションと、特定のタグが付いているポストをスキャンする新しいクエリもセットアップします。

- [Schema (スキーマ)] タブを選択します。
- [スキーマ] ペインの Post タイプを次のように変更して、新しい tags フィールドを追加します。

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
```

```
  downs: Int!
  version: Int!
  tags: [String!]
}
```

- [スキーマ] ペインの Query タイプを次のように変更して、新しい allPostsByTag クエリを追加します。

```
type Query {
  allPostsByTag(tag: String!, count: Int, nextToken: String): PaginatedPosts!
  allPostsByAuthor(author: String!, count: Int, nextToken: String): PaginatedPosts!
  allPost(count: Int, nextToken: String): PaginatedPosts!
  getPost(id: ID): Post
}
```

- [スキーマ] ペインの Mutation タイプを変更して、新しい addTag と removeTag ミューテーションを次のように追加します。

```
type Mutation {
  addTag(id: ID!, tag: String!): Post
  removeTag(id: ID!, tag: String!): Post
  deletePost(id: ID!, expectedVersion: Int): Post
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}
```

- [保存] を選択します。

- 右側のデータ型ペインで、ミューテーション型の新しく作成された allPostsByTag フィールドを見つけて、[アタッチ] を選択します。
- [データソース名] で、[PostDynamoDBTable] を選択します。
- 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "filter": {
      "expression": "contains (tags, :tag)",
        "expressionValues": {
          ":tag": $util.dynamodb.toDynamoDBJson($context.arguments.tag)
        }
    }
    #if( ${context.arguments.count} )
        ,"limit": $util.toJson($context.arguments.count)
    #end
    #if( ${context.arguments.nextToken} )
        ,"nextToken": $util.toJson($context.arguments.nextToken)
    #end
}
```

- 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

- [保存] を選択します。
- 右側のデータ型ペインで、ミューテーション型の新しく作成された addTag フィールドを見つけて、[アタッチ] を選択します。
- [データソース名] で、[PostDynamoDBTable] を選択します。
- 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
```

```
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "ADD tags :tags, version :plusOne",
        "expressionValues" : {
            ":tags" : { "SS": [ $util.toJson($context.arguments.tag) ] },
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
$utils.toJson($context.result)
```

- [保存] を選択します。
- 右側のデータ型ペインで、ミューテーション型の新しく作成された removeTag フィールドを見つけて、[アタッチ] を選択します。
- [データソース名] で、[PostDynamoDBTable] を選択します。
- 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "DELETE tags :tags ADD version :plusOne",
        "expressionValues" : {
            ":tags" : { "SS": [ $util.toJson($context.arguments.tag) ] },
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
$utils.toJson($context.result)
```

- [保存] を選択します。

タグを操作する API の呼び出し

リゾルバーをセットアップしたので、 AWS AppSync は受信 addTag、、removeTagおよび allPostsByTagリクエストを DynamoDBUpdateItem および Scanオペレーションに変換する方法を理解しています。

それを試すには、前のステップで作成したポストのいずれかを選択します。例えば、Nadia が作成したポストを使用しましょう。

- [クエリ] タブを選択します。
- 以下のクエリを [クエリ] ペインに貼り付けます。

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- Nadia のすべてのポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
```

```
    }
  }
}
```

- タイトルが「"The cutest dog in the world"」のポストを使用しましょう。id は後で使用するため書き留めておきます。

では、dog タグを追加してみましょう。

- 以下のミューテーションを [クエリ] ペインに貼り付けます。また、id 引数を、前にメモしておいた値に更新する必要があります。

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- ポストが新しいタグで更新されています。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

次のようにタグを追加できます。

- puppy に変更するように、tag 引数を更新します。

```
mutation addTag {
```

```
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- ポストが新しいタグで更新されています。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

タグを削除することもできます。

- 以下のミューテーションを [クエリ] ペインに貼り付けます。また、id 引数を、前にメモしておいた値に更新する必要があります。

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- ポストが更新され、puppy タグが削除されています。

```
{
  "data": {
```

```
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

タグが付いているすべてのポストを検索することもできます。

- 以下のクエリを [クエリ] ペインに貼り付けます。

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- 次のように dog タグが付いているすべての投稿が返されます。

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
            "dog",
            "puppy"
          ]
        }
      ],
      "nextToken": null
```

```
    }
  }
}
```

リスト型とマップ型の使用

DynamoDB のセット型を使用するだけでなく、DynamoDB のリスト型やマップ型を使用して、複雑なデータを単一のオブジェクトにモデル化することもできます。

ポストにコメントを追加する機能を追加しましょう。これは、DynamoDB 内の Post オブジェクトで、マップ型オブジェクトのリスト型としてモデル化されます。

注 : 実際のアプリケーションでは、独自のテーブル内のコメントをモデル化します。このチュートリアルでは、Post テーブルにコメントを追加しているだけです。

- [Schema (スキーマ)] タブを選択します。
- [Schema (スキーマ)] ペインで、次のように新しい Comment 型を追加します。

```
type Comment {
    author: String!
    comment: String!
}
```

- [スキーマ] ペインの Post タイプを次のように変更して、新しい comments フィールドを追加します。

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
  comments: [Comment!]
}
```

- [スキーマ] ペインの Mutation タイプを次のように変更して、新しい addComment ミューテーションを追加します。

```
type Mutation {
  addComment(id: ID!, author: String!, comment: String!): Post
  addTag(id: ID!, tag: String!): Post
  removeTag(id: ID!, tag: String!): Post
  deletePost(id: ID!, expectedVersion: Int): Post
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}
```

- [保存] を選択します。
- 右側のデータ型ペインで、ミューテーション型の新しく作成された addComment フィールドを見つけて、[アタッチ] を選択します。
- [データソース名] で、[PostDynamoDBTable] を選択します。
- 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "update" : {
    "expression" : "SET comments =
 list_append(if_not_exists(comments, :emptyList), :newComment) ADD version :plusOne",
    "expressionValues" : {
      ":emptyList": { "L" : [] },
      ":newComment" : { "L" : [
```

```
        { "M": {
          "author": $util.dynamodb.toDynamoDBJson($context.arguments.author),
          "comment": $util.dynamodb.toDynamoDBJson($context.arguments.comment)
          }
        }
      ] },
      ":plusOne" : $util.dynamodb.toDynamoDBJson(1)
    }
  }
}
```

この更新式によって、新しいコメントが含まれているリストが、既存の comments リストに追加されます。リストがまだ存在しない場合は作成されます。

- 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
$utils.toJson($context.result)
```

- [保存] を選択します。

コメントを追加する API の呼び出し

リゾルバーをセットアップしたので、 AWS AppSync は受信addCommentリクエストを DynamoDBUpdateItem オペレーションに変換する方法を理解しています。

タグを追加したのと同じポストにコメントを追加して、試してみましょう。

- [クエリ] タブを選択します。
- 以下のクエリを [クエリ] ペインに貼り付けます。

```
mutation addComment {
  addComment(
    id:10
    author: "Steve"
    comment: "Such a cute dog."
  ) {
    id
    comments {
      author
      comment
    }
  }
```

```
}
```

- [Execute query (クエリを実行)] (オレンジ色の再生ボタン) を選択します。
- Nadia のすべてのポストが、クエリペインの右側にある結果ペインに表示されます。これは次のように表示されます。

```
{
  "data": {
    "addComment": {
      "id": "10",
      "comments": [
        {
          "author": "Steve",
          "comment": "Such a cute dog."
        }
      ]
    }
  }
}
```

このリクエストを複数回実行すると、複数のコメントがリストに追加されます。

結論

このチュートリアルでは、 AWS AppSync と GraphQL を使用して DynamoDB 内のポストオブジェクトを操作できる API を構築しました。詳細については、「リゾルバーのマッピングテンプレートリファレンス」を参照してください。

クリーンアップするには、AppSync GraphQL API をコンソールから削除します。

このチュートリアル用に作成した DynamoDB テーブルと IAM ロールを削除するには、以下を実行してAWSAppSyncTutorialForAmazonDynamoDBスタックを削除するか、 AWS CloudFormation コンソールにアクセスしてスタックを削除します。

```
aws cloudformation delete-stack \
    --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

での AWS Lambda リゾルバーの使用 AWS AppSync

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync AWS Lambda で を使用すると、GraphQL フィールドを解決できます。例えば、GraphQL クエリが Amazon Relational Database Service (Amazon RDS) インスタンスを呼び出し、GraphQL のミューテーションを Amazon Kinesis ストリームに書き込むことができます。このセクションでは、GraphQL フィールド処理の呼び出しに応じてビジネスロジックを実行する Lambda 関数を記述する方法について説明します。

Lambda 関数を作成する

以下の例は、Node.js に記述された、ブログ投稿アプリケーションの一部としてブログ投稿に関するさまざまなオペレーションを実行する Lambda 関数を示しています。

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    var posts = {
         "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://
amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR
 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100",
 "downs": "10"},
         "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://
amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups":
 "100", "downs": "10"},
         "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null,
 "content": null, "ups": null, "downs": null },
         "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
         "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} };

    var relatedPosts = {
        "1": [posts['4']],
```

```
        "2": [posts['3'], posts['5']],
        "3": [posts['2'], posts['1']],
        "4": [posts['2'], posts['1']],
        "5": []
    };

    console.log("Got an Invoke Request.");
    switch(event.field) {
        case "getPost":
            var id = event.arguments.id;
            callback(null, posts[id]);
            break;
        case "allPosts":
            var values = [];
            for(var d in posts){
                values.push(posts[d]);
            }
            callback(null, values);
            break;
        case "addPost":
            // return the arguments back
            callback(null, event.arguments);
            break;
        case "addPostErrorWithData":
            var id = event.arguments.id;
            var result = posts[id];
            // attached additional error information to the post
            result.errorMessage = 'Error with the mutation, data has changed';
            result.errorType = 'MUTATION_ERROR';
            callback(null, result);
            break;
        case "relatedPosts":
            var id = event.source.id;
            callback(null, relatedPosts[id]);
            break;
        default:
            callback("Unknown field, unable to resolve" + event.field, null);
            break;
    }
};
```

この Lambda 関数は、ID による投稿の取得、投稿の追加、投稿のリストの取得、および指定した投稿に関連する投稿の取得を行います。

注意 : `event.field` の `switch` ステートメントにより、Lambda 関数は現在解決しているフィールドを確認することができます。

AWS マネジメントコンソールまたは AWS CloudFormation スタックを使用して、この Lambda 関数を作成します。CloudFormation スタックから関数を作成するには、次の AWS Command Line Interface (AWS CLI) コマンドを使用できます。

```
aws cloudformation create-stack --stack-name AppSyncLambdaExample \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/lambda/
LambdaCFTemplate.yaml \
--capabilities CAPABILITY_NAMED_IAM
```

ここから AWS 、アカウントの米国西部 (オレゴン) AWS リージョンで AWS CloudFormation スタックを起動することもできます。

Lambda のデータソースを設定する

Lambda 関数を作成した後、 AWS AppSync コンソールで GraphQL API に移動し、[データソース] タブを選択します。

[データソースの作成] を選択し、[データソース名]としてわかりやすい名前 (**Lambda** など) を入力してから、[データソースタイプ] に対して [AWS Lambda 関数]を選択します。[リージョン] で、関数と同じリージョンを選択します。(提供されている CloudFormation スタックから関数を作成した場合、その関数はおそらく US-WEST-2 にあります)。[関数 ARN] で、Lambda 関数の Amazon リソースネーム (ARN)を選択します。

Lambda 関数を選択したら、新しい AWS Identity and Access Management (IAM) ロール (AWS AppSync が適切なアクセス許可を割り当てるロール) を作成するか、次のインラインポリシーを持つ既存のロールを選択できます。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/
LAMBDA_FUNCTION"
        }
    ]
}
```

また、次のように IAM ロールの AWS AppSync との信頼関係を設定する必要があります。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "appsync.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

GraphQL スキーマを作成する

データソースが Lambda 関数に接続されたので、GraphQL スキーマを作成します。

AWS AppSync コンソールのスキーマエディタから、スキーマが次のスキーマと一致していることを確認します。

```
schema {
    query: Query
    mutation: Mutation
}

type Query {
```

```
    getPost(id:ID!): Post
    allPosts: [Post]
}

type Mutation {
    addPost(id: ID!, author: String!, title: String, content: String, url: String):
 Post!
}

type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    relatedPosts: [Post]
}
```

リゾルバーを設定する

Lambda のデータソースと有効な GraphQL スキーマが登録されたので、リゾルバーを使用して GraphQL フィールドを Lambda のデータソースに接続することができます。

リゾルバーを作成するには、マッピングテンプレートが必要です。マッピングテンプレートの詳細については、「Resolver Mapping Template Overview」を参照してください。

Lambda マッピングテンプレートの詳細については、「Resolver mapping template reference for Lambda」を参照してください。

このステップでは、`getPost(id:ID!): Post`、`allPosts: [Post]`、`addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`、および `Post.relatedPosts: [Post]` の各フィールドで Lambda 関数にリゾルバーをアタッチします。

AWS AppSync コンソールのスキーマエディタから、右側の「 のリゾルバーのアタッチ」を選択しますgetPost(id:ID!): Post。

次に、[アクション] メニューで [ランタイムの更新] を選択し、[ユニットリゾルバー (VTL のみ)] を選択します。

その後、Lambda データソースを選択します。[request mapping template (リクエストマッピングテンプレート)] セクションで、[Invoke And Forward Arguments (呼び出しと引数の転送)] を選択します。

payload オブジェクトを変更し、フィールド名を追加します。テンプレートは次のようになります。

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments":  $utils.toJson($context.arguments)
    }
}
```

[response mapping template (レスポンスマッピングテンプレート)] セクションで、[Return Lambda Result (Lambda 関数の結果を返す)] を選択します。

この場合、ベーステンプレートをそのまま使用します。次のようになります。

```
$utils.toJson($context.result)
```

[保存] を選択します。最初のリゾルバーが正常に追加されました。以下のように、残りのフィールドについてこの操作を繰り返します。

addPost(id: ID!, author: String!, title: String, content: String, url: String): Post! リクエストマッピングテンプレートについて

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "addPost",
        "arguments":  $utils.toJson($context.arguments)
    }
}
```

addPost(id: ID!, author: String!, title: String, content: String, url: String): Post! レスポンスマッピングテンプレートについて

```
$utils.toJson($context.result)
```

allPosts: [Post] リクエストマッピングテンプレートについて

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "allPosts"
    }
}
```

allPosts: [Post] レスポンスマッピングテンプレートについて

```
$utils.toJson($context.result)
```

Post.relatedPosts: [Post] リクエストマッピングテンプレートについて

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "relatedPosts",
        "source":  $utils.toJson($context.source)
    }
}
```

Post.relatedPosts: [Post] レスポンスマッピングテンプレートについて

```
$utils.toJson($context.result)
```

GraphQL API をテストする

これで Lambda 関数が GraphQL リゾルバーに接続されたので、コンソールまたはクライアントアプリケーションを使用してミューテーションまたはクエリが実行できます。

AWS AppSync コンソールの左側でクエリを選択し、次のコードを貼り付けます。

addPost ミューテーション

```
mutation addPost {
```

```
    addPost(
        id: 6
        author: "Author6"
        title: "Sixth book"
        url: "https://www.amazon.com/"
        content: "This is the book is a tutorial for using GraphQL with AWS AppSync."
    ) {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

getPost クエリ

```
query getPost {
    getPost(id: "2") {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

allPosts クエリ

```
query allPosts {
    allPosts {
        id
        author
        title
        content
        url
        ups
        downs
```

```
        relatedPosts {
            id
            title
        }
    }
}
```

エラーを返す

指定したフィールドの解決でエラーが発生する場合があります。 AWS AppSync を使用すると、次のソースからエラーを生成できます。

- リクエストまたはレスポンスのマッピングテンプレート
- Lambda function

マッピングテンプレートからの場合

Velocity Template Language (VTL) テンプレートから `$utils.error` ヘルパーメソッドを使用して故意にエラーを発生させることができます。これは引数として、`errorMessage`、`errorType`、および必要に応じて `data` の各値を含みます。`data` は、エラーの発生時にクライアントに追加のデータを返す場合に利用できます。`data` オブジェクトは GraphQL の最終レスポンスで `errors` に追加されます。

次の例では、`Post.relatedPosts: [Post]` レスポンスマッピングテンプレートでそれを使用する方法を示しています。

```
$utils.error("Failed to fetch relatedPosts", "LambdaFailure", $context.result)
```

これにより、以下のような GraphQL レスポンスが生成されます。

```
{
    "data": {
        "allPosts": [
            {
                "id": "2",
                "title": "Second book",
                "relatedPosts": null
            },
            ...
        ]
```

```
    },
    "errors": [
        {
            "path": [
                "allPosts",
                0,
                "relatedPosts"
            ],
            "errorType": "LambdaFailure",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ],
            "message": "Failed to fetch relatedPosts",
            "data": [
                {
                  "id": "2",
                  "title": "Second book"
                },
                {
                  "id": "1",
                  "title": "First book"
                }
            ]
        }
    ]
}
```

この場合、エラーおよび errorMessage、errorType、data が data.errors[0] オブジェクトに存在するため、allPosts[0].relatedPosts は null になります。

Lambda 関数から

AWS AppSync は、Lambda 関数がスローするエラーも理解します。Lambda プログラミングモデルを使用し、処理されるエラーを発生させることができます。Lambda 関数がエラーをスローした場合、 AWS AppSync は現在のフィールドの解決に失敗します。Lambda から返されたエラーメッセージのみがレスポンスに設定されます。また現在のところ、Lambda 関数からエラーを発生させて、クライアントにエラー関連以外のデータを渡すことはできません。

注: Lambda 関数が未処理のエラーを発生させた場合、 AWS AppSync は Lambda が設定したエラーメッセージを使用します。

以下の Lambda 関数はエラーを発生させます。

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    callback("I fail. Always.");
};
```

これにより、以下のような GraphQL レスポンスが返されます。

```
{
    "data": {
        "allPosts": [
            {
                "id": "2",
                "title": "Second book",
                "relatedPosts": null
            },
            ...
        ]
    },
    "errors": [
        {
            "path": [
                "allPosts",
                0,
                "relatedPosts"
            ],
            "errorType": "Lambda:Handled",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ],
            "message": "I fail. Always."
        }
    ]
}
```

高度なユースケース: バッチ処理

この例の Lambda 関数には、指定した投稿に関連する投稿のリストを返す relatedPosts フィールドが含まれています。この例のクエリでは、Lambda 関数からの allPosts フィールドの呼び出しにより 5 件の投稿が返されます。返された各投稿に対して relatedPosts の解決も指定しているので、relatedPosts フィールドの処理が続けて 5 回呼び出されます。

```
query allPosts {
    allPosts {   // 1 Lambda invocation - yields 5 Posts
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {   // 5 Lambda invocations - each yields 5 posts
            id
            title
        }
    }
}
```

この例では大して意味がないように思われますが、こうした余分な取得処理が積み重なることで、アプリケーションに急速に害をなす可能性があります。

同じクエリで返された関連する Posts について、再度 relatedPosts を取得すると、呼び出しの数は大幅に増加します。

```
query allPosts {
    allPosts {   // 1 Lambda invocation - yields 5 Posts
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {   // 5 Lambda invocations - each yield 5 posts = 5 x 5 Posts
            id
            title
```

```
            relatedPosts {  // 5 x 5 Lambda invocations - each yield 5 posts = 25 x 5
 Posts
                id
                title
                author
            }
        }
    }
}
```

この比較的単純なクエリでは、 AWS AppSync は Lambda 関数を 1 + 5 + 25 = 31 回呼び出します。

これはよくある課題であり、N+1 問題と呼ばれます (この例では、N = 5)。これによりアプリケーションのレイテンシーとコストが増大します。

この問題を解決する 1 つの方法は、同様なフィールドのリゾルバーリクエストを同時にバッチ処理することです。この例では、Lambda 関数は指定された 1 つの投稿に関連する投稿のリストを解決するのではなく、指定された一連の投稿に関連する投稿のリストを解決できます。

これを示すため、`Post.relatedPosts: [Post]` リゾルバーをバッチ処理が有効なリゾルバーに切り替えます。

AWS AppSync コンソールの右側で、既存の`Post.relatedPosts: [Post]`リゾルバーを選択します。リクエストマッピングテンプレートを次のように変更します。

```
{
    "version": "2017-02-28",
    "operation": "BatchInvoke",
    "payload": {
        "field": "relatedPosts",
        "source":  $utils.toJson($context.source)
    }
}
```

`operation` フィールドのみを `Invoke` から `BatchInvoke` に変更します。ペイロードフィールドはテンプレートで指定した内容の配列になっています。この例では、Lambda 関数は、入力として以下を受け取ります。

```
[
    {
        "field": "relatedPosts",
        "source": {
```

```
            "id": 1
        }
    },
    {
        "field": "relatedPosts",
        "source": {
            "id": 2
        }
    },
    ...
]
```

リクエストマッピングテンプレートで BatchInvoke を指定した場合、Lambda 関数はリクエストのリストを受け取り、結果のリストを返します。

具体的には、 AWS AppSync がそれに応じて結果を一致できるように、結果のリストがリクエストペイロードエントリのサイズと順序と一致する必要があります。

このバッチ処理の例では、Lambda 関数は次のように一連の結果を返します。

```
[
    [{"id":"2","title":"Second book"}, {"id":"3","title":"Third book"}],   //
 relatedPosts for id=1
    [{"id":"3","title":"Third book"}]
           // relatedPosts for id=2
]
```

次の Node.js の Lambda 関数は、Post.relatedPosts フィールドに対してこのバッチ処理機能を次のように実行します。

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    var posts = {
         "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://
amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR
 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100",
 "downs": "10"},
         "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://
amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups":
 "100", "downs": "10"},
         "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null,
 "content": null, "ups": null, "downs": null },
```

```
         "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
         "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} };

    var relatedPosts = {
        "1": [posts['4']],
        "2": [posts['3'], posts['5']],
        "3": [posts['2'], posts['1']],
        "4": [posts['2'], posts['1']],
        "5": []
    };

    console.log("Got a BatchInvoke Request. The payload has %d items to resolve.",
 event.length);
    // event is now an array
    var field = event[0].field;
    switch(field) {
        case "relatedPosts":
            var results = [];
            // the response MUST contain the same number
            // of entries as the payload array
            for (var i=0; i< event.length; i++) {
                console.log("post {}", JSON.stringify(event[i].source));
                results.push(relatedPosts[event[i].source.id]);
            }
            console.log("results {}", JSON.stringify(results));
            callback(null, results);
            break;
        default:
            callback("Unknown field, unable to resolve" + field, null);
            break;
    }
};
```

個々のエラーを返す

Lambda 関数から単一のエラーを返せることや、マッピングテンプレートから 1 つのエラーを生成できることは以前の例で説明しました。バッチ処理を呼び出した場合、Lambda 関数からエラーが発生すると、バッチ処理全体が失敗としてフラグ付けされます。これは、データストアとの接続が切れた

場合など、回復不可能なエラーが発生するシナリオでは問題ないかもしれません。ここでは、バッチ処理の一部が成功し、他が失敗した場合、エラーと有効なデータの両方を返すことができます。AWS AppSync はバッチの元のサイズに一致する要素を一覧表示するためにバッチレスポンスを必要とするため、有効なデータを エラーと区別できるデータ構造を定義する必要があります。

例えば、関連する一連の投稿が返されることを Lambda 関数が期待している場合には、data、errorMessage、errorType の各フィールドを任意に含む Response オブジェクトのリストを返します。errorMessage フィールドが存在する場合は、エラーが発生したことを意味します。

次のコードは Lambda 関数の更新方法を示しています。

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    var posts = {
         "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://
amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR
 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100",
 "downs": "10"},
         "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://
amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups":
 "100", "downs": "10"},
         "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null,
 "content": null, "ups": null, "downs": null },
         "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
         "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} };

    var relatedPosts = {
        "1": [posts['4']],
        "2": [posts['3'], posts['5']],
        "3": [posts['2'], posts['1']],
        "4": [posts['2'], posts['1']],
        "5": []
    };

    console.log("Got a BatchInvoke Request. The payload has %d items to resolve.",
 event.length);
    // event is now an array
    var field = event[0].field;
```

```
    switch(field) {
        case "relatedPosts":
            var results = [];
            results.push({ 'data': relatedPosts['1'] });
            results.push({ 'data': relatedPosts['2'] });
            results.push({ 'data': null, 'errorMessage': 'Error Happened', 'errorType':
 'ERROR' });
            results.push(null);
            results.push({ 'data': relatedPosts['3'], 'errorMessage': 'Error Happened
 with last result', 'errorType': 'ERROR' });
            callback(null, results);
            break;
        default:
            callback("Unknown field, unable to resolve" + field, null);
            break;
    }
};
```

この例では、次のレスポンスマッピングテンプレートが Lambda 関数の各項目を解析し、発生するエラーがあればそれを生成します。

```
#if( $context.result && $context.result.errorMessage )
    $utils.error($context.result.errorMessage, $context.result.errorType,
 $context.result.data)
#else
    $utils.toJson($context.result.data)
#end
```

この例では、以下のような GraphQL レスポンスが返されます。

```
{
  "data": {
    "allPosts": [
      {
        "id": "1",
        "relatedPostsPartialErrors": [
          {
            "id": "4",
            "title": "Fourth book"
          }
        ]
      },
      {
```

```
        "id": "2",
        "relatedPostsPartialErrors": [
          {
            "id": "3",
            "title": "Third book"
          },
          {
            "id": "5",
            "title": "Fifth book"
          }
        ]
      },
      {
        "id": "3",
        "relatedPostsPartialErrors": null
      },
      {
        "id": "4",
        "relatedPostsPartialErrors": null
      },
      {
        "id": "5",
        "relatedPostsPartialErrors": null
      }
    ]
  },
  "errors": [
    {
      "path": [
        "allPosts",
        2,
        "relatedPostsPartialErrors"
      ],
      "errorType": "ERROR",
      "locations": [
        {
          "line": 4,
          "column": 9
        }
      ],
      "message": "Error Happened"
    },
    {
      "path": [
```

```
        "allPosts",
        4,
        "relatedPostsPartialErrors"
      ],
      "data": [
        {
          "id": "2",
          "title": "Second book"
        },
        {
          "id": "1",
          "title": "First book"
        }
      ],
      "errorType": "ERROR",
      "locations": [
        {
          "line": 4,
          "column": 9
        }
      ],
      "message": "Error Happened with last result"
    }
  ]
}
```

最大バッチサイズの設定

デフォルトでは、 を使用する場合BatchInvoke、 AWS AppSync は最大 5 つの項目のバッチで Lambda 関数にリクエストを送信します。Lambda リゾルバーの最大バッチサイズを設定できます。

リゾルバーで最大バッチサイズを設定するには、 AWS Command Line Interface () で次のコマンドを使用しますAWS CLI。

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
 relatedPosts \
 --request-mapping-template "<template>" --response-mapping-template "<template>" --
data-source-name "<lambda-datasource>" \
 --max-batch-size X
```

> **Note**
>
> リクエストマッピングテンプレートを提供するときは、バッチ処理を使用する BatchInvoke オペレーションを使用する必要があります。

次のコマンドを使用して、ダイレクト Lambda リゾルバーのバッチ処理を有効にして設定することもできます。

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
 relatedPosts \
 --data-source-name "<lambda-datasource>" \
 --max-batch-size X
```

VTL テンプレートによる最大バッチサイズ設定

VTL のリクエスト内テンプレートがある Lambda リゾルバーの場合、最大バッチサイズは VTL の BatchInvoke オペレーションとして直接指定しない限り、効果がありません。同様に、トップレベルのミューテーションを実行する場合、GraphQL 仕様では並列ミューテーションを順次実行する必要があるため、ミューテーションのバッチ処理は行われません。

例として、次のミューテーションをとりあげます。

```
type Mutation {
    putItem(input: Item): Item
    putItems(inputs: [Item]): [Item]
}
```

最初のミューテーションを使うと、以下のスニペットに示すように 10 個の Items を作成できます。

```
mutation MyMutation {
    v1: putItem($someItem1) {
        id,
        name
    }
    v2: putItem($someItem2) {
        id,
        name
    }
    v3: putItem($someItem3) {
```

```
        id,
        name
    }
    v4: putItem($someItem4) {
        id,
        name
    }
    v5: putItem($someItem5) {
        id,
        name
    }
    v6: putItem($someItem6) {
        id,
        name
    }
    v7: putItem($someItem7) {
        id,
        name
    }
    v8: putItem($someItem8) {
        id,
        name
    }
    v9: putItem($someItem9) {
        id,
        name
    }
    v10: putItem($someItem10) {
        id,
        name
    }
}
```

この例では、Lambda Resolver の最大バッチサイズが 10 に設定されていても、Items は 10 の のグループでバッチ処理されません。代わりに、GraphQL の仕様に従って順番に実行されます。

実際にバッチミューテーションを実行するには、以下の例のように 2 つ目のミューテーションを使うといいでしょう。

```
mutation MyMutation {
    putItems([$someItem1, $someItem2, $someItem3,$someItem4, $someItem5, $someItem6,
    $someItem7, $someItem8, $someItem9, $someItem10]) {
    id,
```

```
    name
    }
}
```

ダイレクト Lambda リゾルバーでのバッチ処理の使用方法については、ダイレクトLambda リゾルバー を参照してください。

AWS AppSync での Amazon OpenSearch Service リゾルバーの使用

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync は、VPC 内に存在しない場合、独自の AWS アカウントでプロビジョニングしたドメインからの Amazon OpenSearch Service の使用をサポートします。ドメインをプロビジョニングした後、データソースを使用してそれに接続できます。この時点で、スキーマにリゾルバーを設定し、クエリ、ミューテーション、サブスクリプションなどの GraphQL 処理を実行することができます。このチュートリアルでは、いくつかの一般的な例を説明します。

詳細については、「OpenSearch 用リゾルバーのマッピングテンプレートリファレンス」を参照してください。

ワンクリックでのセットアップ

Amazon OpenSearch Service を設定して AWS AppSync で GraphQL エンドポイントを自動的にセットアップするには、この AWS CloudFormation テンプレートを使用します。

AWS CloudFormation デプロイが完了したら、GraphQL クエリとミューテーションの実行に直接スキップできます。

OpenSearch Service ドメインを作成する

このチュートリアルを開始するには、既存の OpenSearch Service ドメインが必要です。ドメインがない場合は、次のサンプルが使用できます。OpenSearch Service ドメインが作成されるまで、AWS AppSync データソースとの統合に進むまでに最大 15 分かかる場合があります。

```
aws cloudformation create-stack --stack-name AppSyncOpenSearch \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/elasticsearch/
ESResolverCFTemplate.yaml \
--parameters ParameterKey=OSDomainName,ParameterValue=ddtestdomain
 ParameterKey=Tier,ParameterValue=development \
--capabilities CAPABILITY_NAMED_IAM
```

AWS アカウントの米国西部 2 (オレゴン) リージョンで次の AWS CloudFormation スタックを起動できます。

OpenSearch Service のデータソースの設定

OpenSearch Service ドメインを作成したら、 AWS AppSync GraphQL API に移動し、データソースタブを選択します。[New (新規)] を選択して、データソースに「oss」などの分かりやすい名前を入力します。次に、[データソースタイプ] に [Amazon Opensearch ドメイン] を選択し、適切なリージョンを選択します。OpenSearch Service ドメインがリストに表示されます。選択したら、新しいロールを作成し、 AWS AppSync がロールに適切なアクセス許可を割り当てるか、次のインラインポリシーを持つ既存のロールを選択できます。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1234234",
            "Effect": "Allow",
            "Action": [
                "es:ESHttpDelete",
                "es:ESHttpHead",
```

```
                "es:ESHttpGet",
                "es:ESHttpPost",
                "es:ESHttpPut"
            ],
            "Resource": [
                "arn:aws:es:us-east-1:ACCOUNTNUMBER:domain/democluster/*"
            ]
        }
    ]
}
```

また、そのロールに対して AWS AppSync との信頼関係を設定する必要があります。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "appsync.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

さらに、OpenSearch Service ドメインには独自のアクセスポリシーがあります。これは Amazon OpenSearch Service コンソールを使用して変更できます。OpenSearch Service ドメインの適切なアクションとリソースを用いて、以下のように同じようなポリシーを追加する必要があります。Principal は AppSync のデータソースのロールです。これをコンソールに作成させる場合には、IAM コンソールに表示されるようにします。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:role/service-role/
APPSYNC_DATASOURCE_ROLE"
            },
            "Action": [
                "es:ESHttpDelete",
                "es:ESHttpHead",
                "es:ESHttpGet",
                "es:ESHttpPost",
                "es:ESHttpPut"
            ],
            "Resource": "arn:aws:es:us-east-1:ACCOUNTNUMBER:domain/DOMAIN_NAME/*"
        }
    ]
}
```

リゾルバーを接続する

これで、データソースが OpenSearch Service ドメインに接続されたので、リゾルバーを使用して、以下の例のようにデータソースを GraphQL スキーマに接続することができます。

```
 schema {
   query: Query
   mutation: Mutation
 }

 type Query {
   getPost(id: ID!): Post
   allPosts: [Post]
 }

 type Mutation {
   addPost(id: ID!, author: String, title: String, url: String, ups: Int, downs: Int,
 content: String): AWSJSON
 }

type Post {
  id: ID!
  author: String
  title: String
```

```
  url: String
  ups: Int
  downs: Int
  content: String
}
...
```

Post フィールドを含むユーザー定義の id 型があることに注意してください。次の例では、この型を OpenSearch Service ドメインに配置するプロセス (自動化可能) があることを前提としています。このプロセスでは、/post/_doc へのルートパスをマッピングします。post はインデックスです。このルートパスから、個々のドキュメントの検索、/id/post* によるワイルドカード検索、/post/_search の パスを使用した複数ドキュメントの検索が行えます。例えば、同じインデックス user でインデックスが付けられた User という別の型がある場合は、/user/_search の パスを使用して複数ドキュメントが検索できます。

AWS AppSync コンソールのスキーマエディタから、前述のPostsスキーマを変更してsearchPostsクエリを含めます。

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

スキーマを保存します。右側の [searchPosts] で、[Attach resolver (リゾルバーをアタッチ)] を選択します。[アクション] メニューで [ランタイムの更新] を選択し、[ユニットリゾルバー (VTL のみ)] を選択します。次に、OpenSearch Service サービスのデータソースを選択します。[request mapping template (リクエストマッピングテンプレート)] セクションで、ドロップダウンから [Query posts (投稿のクエリ)] を選択し、基本テンプレートを取得します。path を /post/_search に変更します。次のようになります。

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/post/_search",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "from":0,
```

```
            "size":50
        }
    }
}
```

これは、前述のスキーマの post フィールドの下に OpenSearch Service でインデックス化されたドキュメントがあることを前提としています。データ構造が異なる場合は、更新が必要です。

OpenSearch Service クエリからデータの結果を受信し、変換して GraphQL に渡す場合には、[レスポンスマッピングテンプレート] セクションで適切な _source フィルタを指定する必要があります。以下のテンプレートを使用してください。

```
[
    #foreach($entry in $context.result.hits.hits)
    #if( $velocityCount > 1 ) , #end
    $utils.toJson($entry.get("_source"))
    #end
]
```

検索を変更する

前述のリクエストマッピングテンプレートは、すべてのレコードに簡単なクエリを実行します。今、特定の筆者で検索する場合を考えます。また、その筆者を、GraphQL クエリに定義した引数にするとします。 AWS AppSync コンソールのスキーマエディタで、allPostsByAuthorクエリを追加します。

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

その後、[リゾルバーをアタッチ] を選択し、OpenSearch Service データソースを選択します。ただし、[レスポンスマッピングテンプレート] には次の例を使用します。

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/post/_search",
```

```
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "from":0,
            "size":50,
            "query":{
                "match" :{
                    "author": $util.toJson($context.arguments.author)
                }
            }
        }
    }
}
```

body には author フィールドを含む term クエリが含まれていることに注意してください。これは引数としてクライアントから渡されます。任意で、標準テキストなどの事前に入力された情報を追加したり、他のユーティリティを使用したりすることもできます。

このリゾルバーを使用している場合、以前の例と同じ情報をレスポンスマッピングテンプレートに入力します。

OpenSearch Service へのデータの追加

GraphQL ミューテーションの結果、OpenSearch Service ドメインにデータの追加が必要になる場合があります。これは検索やその他の目的に非常に役立ちます。GraphQL サブスクリプションを使用してデータをリアルタイムで作成できるため、これは OpenSearch Service ドメインのデータの更新をクライアントに通知するメカニズムとして動作します。

AWS AppSync コンソールのスキーマページに戻り、addPost()ミューテーションのリゾルバーをアタッチを選択します。再度 OpenSearch Service データソースを選択し、Posts スキーマに次のレスポンスマッピングテンプレートを使用します。

```
{
    "version":"2017-02-28",
    "operation":"PUT",
    "path": $util.toJson("/post/_doc/$context.arguments.id"),
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
```

```
            "id": $util.toJson($context.arguments.id),
            "author": $util.toJson($context.arguments.author),
            "ups": $util.toJson($context.arguments.ups),
            "downs": $util.toJson($context.arguments.downs),
            "url": $util.toJson($context.arguments.url),
            "content": $util.toJson($context.arguments.content),
            "title": $util.toJson($context.arguments.title)
        }
    }
}
```

以前と同様、これはデータ構造の一例です。別のフィールド名またはインデックスがある場合は、必要に応じて path と body を更新します。この例では、GraphQL ミューテーションの引数からテンプレートを受け取るために $context.arguments を使用する方法も示されています。

次に進む前に、以下のレスポンスマッピングテンプレートを使用してください。これにより、ミューテーション操作の結果またはエラー情報が出力として返されます。

```
#if($context.error)
    $util.toJson($ctx.error)
#else
    $util.toJson($context.result)
#end
```

単一のドキュメントの取得

最後に、スキーマ内のgetPost(id:ID)クエリを使用して個々のドキュメントを返す場合は、 AWS AppSync コンソールのスキーマエディタでこのクエリを検索し、リゾルバーをアタッチを選択します。再度 OpenSearch Service データソースを選択し、次のマッピングテンプレートを使用します。

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path": $util.toJson("post/_doc/$context.arguments.id"),
    "params":{
        "headers":{},
        "queryString":{},
        "body":{}
    }
}
```

上記の path では id 引数の body が空であるため、単一のドキュメントが返ります。ただし、リストではなく単一の項目が返るため、次のレスポンスマッピングテンプレートを使用する必要があります。

```
$utils.toJson($context.result.get("_source"))
```

クエリとミューテーションを実行する

これで、OpenSearch Service ドメインに対して GraphQL 処理が実行できます。 AWS AppSync コンソールのクエリタブに移動し、新しいレコードを追加します。

```
mutation addPost {
    addPost (
        id:"12345"
        author: "Fred"
        title: "My first book"
        content: "This will be fun to write!"
        url: "publisher website",
        ups: 100,
        downs:20
       )
}
```

ミューテーションの結果が右側に表示されます。同様に、OpenSearch Service ドメインに対して searchPosts クエリが実行できます。

```
query searchPosts {
    searchPosts {
        id
        title
        author
        content
    }
}
```

ベストプラクティス

- OpenSearch Service はプライマリデータベースとしてではなく、データのクエリ発行のために使用します。「GraphQL リゾルバーを組み合わせる」で説明したように、Amazon DynamoDB と組み合わせて OpenSearch Service を使用する場合があります。

- ドメインへのアクセスを許可するには、 AWS AppSync サービスロールにクラスターへのアクセスを許可します。
- 開発中は、最小限のコストのクラスターを使用して小規模で開始し、その後、本稼働への移行時に高可用性 (HA) を備えた大規模なクラスターへと移行することができます。

AWS AppSync でのローカルリゾルバーの使用

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync では、サポートされているデータソース (AWS Lambda、Amazon DynamoDB、または Amazon OpenSearch Service) を使用してさまざまなオペレーションを実行できます。ただし、特定のシナリオでは、サポートされているデータソースに対する呼び出しの必要がないことがあります。

そのような場合は、ローカルリゾルバーが役立ちます。リモートのデータソースを呼び出すのではなく、ローカルリゾルバーはリクエストマッピングテンプレートの結果をレスポンスマッピングテンプレートに転送します。フィールド解決は AWS AppSync を離れません。

ローカルリゾルバーは、いくつかのユースケースで便利です。特によく使用されるのは、データソースの呼び出しをトリガーせずに通知を発行する場合です。そのユースケースの例を示すために、ユーザーが互いに呼び出すことができるページングアプリケーションを構築しましょう。この例では、サブスクリプションを活用します。サブスクリプションに慣れていない場合は、「リアルタイムデータ」チュートリアルを参照してください。

ページングアプリケーションの作成

このページングアプリケーションでは、クライアントは受信箱をサブスクライブでき、他のクライアントにページを送信できます。各ページにはメッセージが含まれています。スキーマは次のとおりです。

```
schema {
    query: Query
    mutation: Mutation
```

```
    subscription: Subscription
}

type Subscription {
    inbox(to: String!): Page
    @aws_subscribe(mutations: ["page"])
}

type Mutation {
    page(body: String!, to: String!): Page!
}

type Page {
    from: String
    to: String!
    body: String!
    sentAt: String!
}

type Query {
    me: String
}
```

Mutation.page フィールドにリゾルバーをアタッチしましょう。[スキーマ] ペインで、右側のパネルのフィールド定義の横にある [Attach Resolver (リゾルバーのアタッチ)] をクリックします。None 型の新しいデータソースを作成して、PageDataSource という名前を付けます。

リクエストマッピングテンプレートで、次のように入力します。

```
{
  "version": "2017-02-28",
  "payload": {
    "body": $util.toJson($context.arguments.body),
    "from": $util.toJson($context.identity.username),
    "to":  $util.toJson($context.arguments.to),
    "sentAt": "$util.time.nowISO8601()"
  }
}
```

レスポンスマッピングテンプレートで、デフォルトの [Forward the result (結果を転送)] を選択します。リゾルバーを保存します。これで、アプリケーションの準備ができたので、ページングしましょう。

ページの送信およびサブスクライブ

ページを受信するクライアントで、まず受信箱をサブスクライブする必要があります。

[Queries (クエリ)] ペインで inbox のサブスクリプションを実行しましょう。

```
subscription Inbox {
    inbox(to: "Nadia") {
        body
        to
        from
        sentAt
    }
}
```

Nadia は、Mutation.page ミューテーションが呼び出されるたびにページを受け取ります。ミューテーションを実行することで、このミューテーションを呼び出しましょう。

```
mutation Page {
    page(to: "Nadia", body: "Hello, World!") {
        body
        to
        from
        sentAt
    }
}
```

ここまで、 AWS AppSync を離れずにページを送信して受信することで、ローカルリゾルバーの使用をデモンストレーションしました。

AWS AppSyncでの GraphQL リゾルバーの組み合わせ

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちらにある APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

GraphQL スキーマのリゾルバーとフィールドには、非常に柔軟性の高い 1 対 1 の関係があります。データソースはスキーマとは独立にリゾルバーに設定されるため、ニーズに応じてスキーマ上で組み

合わせやマッチングを行い、さまざまなデータソースを使用して GraphQL 型の解決や操作が行えます。

次のシナリオ例では、スキーマ内のデータソースを混在させて一致させる方法を示します。開始する前に、前のチュートリアルで説明したように AWS Lambda、、Amazon DynamoDB、および Amazon OpenSearch Service のデータソースとリゾルバーの設定に精通しておくことをお勧めします。

スキーマの例

次のスキーマには、3 つの `Query` 処理と 3 つの `Mutation` 処理を含む `Post` という型が定義されています。

```
type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
}

type Query {
    allPost: [Post]
    getPost(id: ID!): Post
    searchPosts: [Post]
}

type Mutation {
    addPost(
        id: ID!,
        author: String!,
        title: String,
        content: String,
        url: String
    ): Post
    updatePost(
        id: ID!,
        author: String!,
        title: String,
        content: String,
```

```
        url: String,
        ups: Int!,
        downs: Int!,
        expectedVersion: Int!
    ): Post
    deletePost(id: ID!): Post
}
```

この例では、合計 6 つのリゾルバーをアタッチします。考えられる方法の 1 つは、これらをすべて Posts という Amazon DynamoDB テーブルから取得することです。この場合、「DynamoDB のリゾルバーのマッピングテンプレートリファレンス」で説明されているように、AllPosts はスキャンを実行し、searchPosts はクエリを実行します。ただし、Lambda または OpenSearch Service を使用してこれらの GraphQL クエリに解決させるなど、ビジネスニーズに応じて他の方法も使用できます。

リゾルバーを使用してデータを変更する

DynamoDB (または Amazon Aurora) などのデータベースからクライアントに、一部の属性を変更した結果を返したい場合があります。クライアントでのタイムスタンプの相違などによるデータ型の成形が必要な場合や、後方互換性を確保するための処理が必要な場合などです。以下では、分かりやすい例として、GraphQL リゾルバーが呼び出されるたびに投稿にランダムな番号を割り当て、ブログの投稿に対する賛成または反対を操作する AWS Lambda 関数を示しています。

```
'use strict';
const doc = require('dynamodb-doc');
const dynamo = new doc.DynamoDB();

exports.handler = (event, context, callback) => {
    const payload = {
        TableName: 'Posts',
        Limit: 50,
        Select: 'ALL_ATTRIBUTES',
    };

    dynamo.scan(payload, (err, data) => {
        const result = { data: data.Items.map(item =>{
            item.ups = parseInt(Math.random() * (50 - 10) + 10, 10);
            item.downs = parseInt(Math.random() * (20 - 0) + 0, 10);
            return item;
        }) };
        callback(err, result.data);
```

```
    });
};
```

これは完全に有効な Lambda 関数であり、GraphQL スキーマの `AllPosts` フィールドにアタッチできるため、すべての結果を返すクエリが、賛成/反対に対するランダムな番号を受け取ります。

DynamoDB と OpenSearch Service

一部のアプリケーションでは、DynamoDB に対してミューテーションまたは簡単な検索クエリを実行し、バックグラウンドプロセスでドキュメントを OpenSearch Service に転送する場合があります。その後、`searchPosts` リゾルバーを OpenSearch Service データソースに単純にアタッチし、GraphQL クエリを使用して、(DynamoDB のデータからの) 検索結果を返します。これは、キーワードやあいまいワードによる検索、地理空間検索などの高度な検索処理をアプリケーションに追加する場合に非常に役立ちます。DynamoDB からのデータ転送は、ETL プロセスを使用して、または Lambda を使用した DynamoDB からのストリームを使用して行うことができます。 AWS アカウントの米国西部 2 (オレゴン) リージョンで次の AWS CloudFormation スタックを使用して、この完全な例を起動できます。

この例では、スキーマは以下のような DynamoDB リゾルバーを使用して投稿を追加します。

```
mutation add {
    putPost(author:"Nadia"
        title:"My first post"
        content:"This is some test content"
        url:"https://aws.amazon.com/appsync/"
    ){
        id
        title
    }
}
```

これにより、データが DynamoDB に書き込まれます。その後、Lambda を介してデータが Amazon OpenSearch Service に転送されます。ここではすべての投稿がさまざまなフィールドについて検索できます。たとえば、データが Amazon OpenSearch Service にあるため、以下のように空白を含む任意のテキストを使用して、author フィールドまたは content フィールドを検索できます。

```
query searchName{
    searchAuthor(name:"   Nadia   "){
```

```
        id
        title
        content
    }
}

query searchContent{
    searchContent(text:"test"){
        id
        title
        content
    }
}
```

データは DynamoDB に直接書き込まれるため、`allPosts{...}` クエリと `singlePost{...}` クエリを使用して、テーブルに対するリストまたは項目の検索処理が効率的に実行できます。このスタックでは、DynamoDB ストリームに対して次のコード例を使用します。

注意 : ここで示しているのは一例にすぎません。

```
var AWS = require('aws-sdk');
var path = require('path');
var stream = require('stream');

var esDomain = {
    endpoint: 'https://opensearch-domain-name.REGION.es.amazonaws.com',
    region: 'REGION',
    index: 'id',
    doctype: 'post'
};

var endpoint = new AWS.Endpoint(esDomain.endpoint)
var creds = new AWS.EnvironmentCredentials('AWS');

function postDocumentToES(doc, context) {
    var req = new AWS.HttpRequest(endpoint);

    req.method = 'POST';
    req.path = '/_bulk';
    req.region = esDomain.region;
    req.body = doc;
    req.headers['presigned-expires'] = false;
    req.headers['Host'] = endpoint.host;
```

```
    // Sign the request (Sigv4)
    var signer = new AWS.Signers.V4(req, 'es');
    signer.addAuthorization(creds, new Date());

    // Post document to ES
    var send = new AWS.NodeHttpClient();
    send.handleRequest(req, null, function (httpResp) {
        var body = '';
        httpResp.on('data', function (chunk) {
            body += chunk;
        });
        httpResp.on('end', function (chunk) {
            console.log('Successful', body);
            context.succeed();
        });
    }, function (err) {
        console.log('Error: ' + err);
        context.fail();
    });
}

exports.handler = (event, context, callback) => {
    console.log("event => " + JSON.stringify(event));
    var posts = '';

    for (var i = 0; i < event.Records.length; i++) {
        var eventName = event.Records[i].eventName;
        var actionType = '';
        var image;
        var noDoc = false;
        switch (eventName) {
            case 'INSERT':
                actionType = 'create';
                image = event.Records[i].dynamodb.NewImage;
                break;
            case 'MODIFY':
                actionType = 'update';
                image = event.Records[i].dynamodb.NewImage;
                break;
            case 'REMOVE':
            actionType = 'delete';
                image = event.Records[i].dynamodb.OldImage;
                noDoc = true;
```

```
                break;
        }

        if (typeof image !== "undefined") {
            var postData = {};
            for (var key in image) {
                if (image.hasOwnProperty(key)) {
                    if (key === 'postId') {
                        postData['id'] = image[key].S;
                    } else {
                        var val = image[key];
                        if (val.hasOwnProperty('S')) {
                            postData[key] = val.S;
                        } else if (val.hasOwnProperty('N')) {
                            postData[key] = val.N;
                        }
                    }
                }
            }

            var action = {};
            action[actionType] = {};
            action[actionType]._index = 'id';
            action[actionType]._type = 'post';
            action[actionType]._id = postData['id'];
            posts += [
                JSON.stringify(action),
            ].concat(noDoc?[]:[JSON.stringify(postData)]).join('\n') + '\n';
        }
    }
    console.log('posts:',posts);
    postDocumentToES(posts, context);
};
```

DynamoDB ストリームを使用し、id のプライマリキーを使用してこれを DynamoDB テーブルにアタッチできます。DynamoDB のソースへの変更は OpenSearch Service ドメインに転送されます。この設定の詳細については、DynamoDB Streams のドキュメントを参照してください。

AWS AppSync での DynamoDB バッチオペレーションの使用

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync は、1 つのリージョン内の 1 つ以上のテーブルで Amazon DynamoDB バッチオペレーションの使用をサポートしています。サポートされている処理は、`BatchGetItem`、`BatchPutItem`、および `BatchDeleteItem` です。 AWS AppSync でこれらの機能を使用すると、次のようなタスクを実行できます。

- 単一クエリでキーのリストを渡し、テーブルからの結果を返す
- 単一クエリで 1 つ以上のテーブルからレコードを読み取る
- 1 つ以上のテーブルに一連のレコードを書き込む
- 関連のある複数のテーブルのレコードを状況に応じて書き込みまたは削除

AWS AppSync で DynamoDB でバッチオペレーションを使用することは、バックエンドオペレーションとテーブル構造について、もう少し詳しく考え、知識を得るための高度な手法です。さらに、AWS AppSync のバッチオペレーションには、バッチ処理されていないオペレーションと 2 つの主な違いがあります。

- データソースのロールには、リゾルバーがアクセスするすべてのテーブルについてアクセス許可が必要です。
- リゾルバーのテーブル仕様はマッピングテンプレートに含まれます。

アクセス許可

他のリゾルバーと同様に、 AWS AppSync でデータソースを作成し、ロールを作成するか、既存のロールを使用する必要があります。バッチ処理では DynamoDB テーブルごとに異なるアクセス許可が必要であるため、読み取りまたは書き込みアクションのために設定されたアクセス許可がロールに必要です。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:us-east-1:account:table/TABLENAME",
                "arn:aws:dynamodb:region:account:table/TABLENAME/*"
            ]
        }
    ]
}
```

注: ロールは AWS AppSync のデータソースに関連付けられ、フィールドのリゾルバーはデータソースに対して呼び出されます。DynamoDB に対して取得するよう設定されたデータソースは、設定を単純にするために 1 つのテーブルのみ指定されます。したがって、単一のリゾルバーで複数のテーブルに対してバッチ処理を実行する場合、これはより高度なタスクであり、リゾルバーが利用するすべてのテーブルへのアクセスをデータソースのロールに許可する必要があります。これは、上記の IAM ポリシーの Resource フィールドで行われます。テーブルに対してバッチを呼び出すためのテーブル設定は、リゾルバーのテンプレートで行います。これについては以下で説明します。

データソース

分かりやすくするために、このチュートリアルではすべてのリゾルバーに同じデータソースを使用します。[Data sources (データソース)] タブで、新しい DynamoDB データソースを作成し、BatchTutorialという名前を付けます。テーブル名はバッチ処理のリクエストマッピングテンプレートで指定するので、何でも構いません。ここでは、テーブル名に `empty` を使用しています。

このチュートリアルでは、次のインラインポリシーを含むロールが使用できます。

JSON

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:us-east-1:account:table/Posts",
                "arn:aws:dynamodb:region:account:table/Posts/*",
                "arn:aws:dynamodb:us-east-1:account:table/locationReadings",
                "arn:aws:dynamodb:region:account:table/locationReadings/*",
                "arn:aws:dynamodb:us-east-1:account:table/temperatureReadings",
                "arn:aws:dynamodb:region:account:table/temperatureReadings/*"
            ]
        }
    ]
}
```

1 つのテーブルのバッチ処理

> **Warning**
>
> BatchPutItem および BatchDeleteItem は、競合の検出と解決とともに使用する場合はサポートされていません。これらの設定は、起こり得るエラーを防ぐために無効にする必要があります。

以下の例では、Posts という名前の単一のテーブルがあり、バッチ処理を使用してこれに項目を追加または削除するとします。次のスキーマを使用し、クエリには何もせずに ID のリストを渡します。

```
type Post {
    id: ID!
    title: String
}

input PostInput {
    id: ID!
    title: String
}
```

```
type Query {
    batchGet(ids: [ID]): [Post]
}

type Mutation {
    batchAdd(posts: [PostInput]): [Post]
    batchDelete(ids: [ID]): [Post]
}

schema {
    query: Query
    mutation: Mutation
}
```

次のリクエストマッピングテンプレートを使用して batchAdd() フィールドにリゾルバーをアタッチします。これは自動的に GraphQL の input PostInput 型に各項目を渡し、マップを作成します。このマップは BatchPutItem の処理に必要になります。

```
#set($postsdata = [])
#foreach($item in ${ctx.args.posts})
    $util.qr($postsdata.add($util.dynamodb.toMapValues($item)))
#end

{
    "version" : "2018-05-29",
    "operation" : "BatchPutItem",
    "tables" : {
        "Posts": $utils.toJson($postsdata)
    }
}
```

このケースでは、レスポンスマッピングテンプレートは単純に実行されますが、次のようにテーブル名が ..data.Posts として context オブジェクトに追加されています。

```
$util.toJson($ctx.result.data.Posts)
```

次に、 AWS AppSync コンソールのクエリページに移動し、次の batchAdd ミューテーションを実行します。

```
mutation add {
```

```
    batchAdd(posts:[{
            id: 1 title: "Running in the Park"},{
            id: 2 title: "Playing fetch"
        }]){
            id
            title
    }
}
```

画面に結果が表示されるので、DynamoDB コンソールを使用して、両方の値がPostsテーブルに書き込まれたことを個別に確認できます。

次に、以下のリクエストマッピングテーブルを使用して batchGet() フィールドにリゾルバーをアタッチします。これは自動的に GraphQL の ids:[] 型に各項目を渡し、マップを作成します。このマップは BatchGetItem の処理に必要になります。

```
#set($ids = [])
#foreach($id in ${ctx.args.ids})
    #set($map = {})
    $util.qr($map.put("id", $util.dynamodb.toString($id)))
    $util.qr($ids.add($map))
#end

{
    "version" : "2018-05-29",
    "operation" : "BatchGetItem",
    "tables" : {
        "Posts": {
            "keys": $util.toJson($ids),
            "consistentRead": true,
            "projection" : {
                "expression" : "#id, title",
                "expressionNames" : { "#id" : "id"}
                }
        }
    }
}
```

再度、レスポンスマッピングテンプレートが単純に実行されますが、ここでもテーブル名が ..data.Posts として context オブジェクトに追加されます。

```
$util.toJson($ctx.result.data.Posts)
```

次に、 AWS AppSync コンソールのクエリページに戻り、次の batchGet Query を実行します。

```
query get {
    batchGet(ids:[1,2,3]){
        id
        title
    }
}
```

これは、以前に追加した 2 つの id 値の結果を返します。値が null の id に対して 3 値が返っていることに注意してください。これは、その値に対応するレコードが Posts テーブルにまだないためです。また、 AWS AppSync はクエリに渡されたキーと同じ順序で結果を返します。これは、 AWS AppSync がユーザーに代わって実行する追加機能です。したがって、batchGet(ids:[1,3,2) に変更すると、順番が代わります。どの id で null 値が返されたのかもこれで分かります。

最後に、以下のリクエストマッピングテンプレートを使用して batchDelete() フィールドにリゾルバーをアタッチします。これは自動的に GraphQL の ids:[] 型に各項目を渡し、マップを作成します。このマップは BatchGetItem の処理に必要になります。

```
#set($ids = [])
#foreach($id in ${ctx.args.ids})
    #set($map = {})
    $util.qr($map.put("id", $util.dynamodb.toString($id)))
    $util.qr($ids.add($map))
#end

{
    "version" : "2018-05-29",
    "operation" : "BatchDeleteItem",
    "tables" : {
        "Posts": $util.toJson($ids)
    }
}
```

再度、レスポンスマッピングテンプレートが単純に実行されますが、ここでもテーブル名が ..data.Posts として context オブジェクトに追加されます。

```
$util.toJson($ctx.result.data.Posts)
```

次に、 AWS AppSync コンソールのクエリページに戻り、次の batchDelete ミューテーションを実行します。

```
mutation delete {
    batchDelete(ids:[1,2]){ id }
}
```

id が 1 と 2 のレコードが削除されます。以前に使用した batchGet() クエリを再実行すると、これらは null を返します。

複数テーブルのバッチ処理

> **Warning**
>
> BatchPutItem および BatchDeleteItem は、競合の検出と解決とともに使用する場合はサポートされていません。これらの設定は、起こり得るエラーを防ぐために無効にする必要があります。

AWS AppSync では、テーブル間でバッチオペレーションを実行することもできます。より複雑なアプリケーションを作成しましょう。Pet Health アプリを作成するとします。これは、センサーによりペットの場所と体温を報告します。センサーは電池により動作し、数分ごとにネットワークへの接続を試みます。センサーが接続を確立すると、その読み取り値が AWS AppSync API に送信されます。その後、データの分析がトリガーされ、ペットの飼い主にダッシュボードが表示されます。センサーとバックエンドのデータストア間のやり取りの作成について考えてみましょう。

前提条件として、まず 2 つの DynamoDB テーブルを作成します。locationReadings はセンサーの温度の読み取り値を格納し、temperatureReadings はセンサーの位置の読み取り値を格納します。両方のテーブルで同じプライマリキー構造体を共有します。sensorId（String）はパーティションキーであり、timestamp（String）がソートキーです。

以下の GraphQL スキーマを使用しましょう。

```
type Mutation {
    # Register a batch of readings
    recordReadings(tempReadings: [TemperatureReadingInput], locReadings:
 [LocationReadingInput]): RecordResult
    # Delete a batch of readings
    deleteReadings(tempReadings: [TemperatureReadingInput], locReadings:
 [LocationReadingInput]): RecordResult
}

type Query {
```

```
    # Retrieve all possible readings recorded by a sensor at a specific time
    getReadings(sensorId: ID!, timestamp: String!): [SensorReading]
}

type RecordResult {
    temperatureReadings: [TemperatureReading]
    locationReadings: [LocationReading]
}

interface SensorReading {
    sensorId: ID!
    timestamp: String!
}

# Sensor reading representing the sensor temperature (in Fahrenheit)
type TemperatureReading implements SensorReading {
    sensorId: ID!
    timestamp: String!
    value: Float
}

# Sensor reading representing the sensor location (lat,long)
type LocationReading implements SensorReading {
    sensorId: ID!
    timestamp: String!
    lat: Float
    long: Float
}

input TemperatureReadingInput {
    sensorId: ID!
    timestamp: String
    value: Float
}

input LocationReadingInput {
    sensorId: ID!
    timestamp: String
    lat: Float
    long: Float
}
```

BatchPutItem - センサーの読み取り値の記録

センサーは、インターネットに接続後、読み取り値を送信できる必要があります。GraphQL の Mutation.recordReadings フィールドは、このために使用される API です。リゾルバーをアタッチし、API を有効にしましょう。

[Mutation.recordReadings] フィールドの横にある [アタッチ] を選択します。次の画面で、チュートリアルの最初で作成したのと同じ BatchTutorial データソースを選択します。

次のリクエストマッピングテンプレートを追加しましょう。

リクエストマッピングテンプレート

```
## Convert tempReadings arguments to DynamoDB objects
#set($tempReadings = [])
#foreach($reading in ${ctx.args.tempReadings})
    $util.qr($tempReadings.add($util.dynamodb.toMapValues($reading)))
#end

## Convert locReadings arguments to DynamoDB objects
#set($locReadings = [])
#foreach($reading in ${ctx.args.locReadings})
    $util.qr($locReadings.add($util.dynamodb.toMapValues($reading)))
#end

{
    "version" : "2018-05-29",
    "operation" : "BatchPutItem",
    "tables" : {
        "locationReadings": $utils.toJson($locReadings),
        "temperatureReadings": $utils.toJson($tempReadings)
    }
}
```

ご覧のように、BatchPutItem 処理により複数のテーブルが指定できます。

次のレスポンスマッピングテンプレートを使用しましょう。

レスポンスマッピングテンプレート

```
## If there was an error with the invocation
## there might have been partial results
#if($ctx.error)
```

```
    ## Append a GraphQL error for that field in the GraphQL response
    $utils.appendError($ctx.error.message, $ctx.error.message)
#end
## Also returns data for the field in the GraphQL response
$utils.toJson($ctx.result.data)
```

バッチ処理では、呼び出しからエラーと結果の両方が返る可能性があります。その場合は、任意に追加のエラー処理を実行できます。

注意: $utils.appendError() の使用法は $util.error() と同様ですが、マッピングテンプレートの評価を中断しないという違いがあります。代わりに、エラーが発生したことをフィールドで通知します。ただし、テンプレートを評価して、データを引き続き呼び出し元に返すことも可能です。アプリケーションで部分的な結果を返す必要がある場合は、$utils.appendError() を使用することを推奨します。

リゾルバーを保存し、 AWS AppSync コンソールのクエリページに移動します。センサーの読み取り値をいくつか送信してみましょう!

次のミューテーションを実行します。

```
mutation sendReadings {
  recordReadings(
    tempReadings: [
      {sensorId: 1, value: 85.5, timestamp: "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, value: 85.7, timestamp: "2018-02-01T17:21:06.000+08:00"},
      {sensorId: 1, value: 85.8, timestamp: "2018-02-01T17:21:07.000+08:00"},
      {sensorId: 1, value: 84.2, timestamp: "2018-02-01T17:21:08.000+08:00"},
      {sensorId: 1, value: 81.5, timestamp: "2018-02-01T17:21:09.000+08:00"}
    ]
    locReadings: [
      {sensorId: 1, lat: 47.615063, long: -122.333551, timestamp:
 "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, lat: 47.615163, long: -122.333552, timestamp:
 "2018-02-01T17:21:06.000+08:00"}
      {sensorId: 1, lat: 47.615263, long: -122.333553, timestamp:
 "2018-02-01T17:21:07.000+08:00"}
      {sensorId: 1, lat: 47.615363, long: -122.333554, timestamp:
 "2018-02-01T17:21:08.000+08:00"}
      {sensorId: 1, lat: 47.615463, long: -122.333555, timestamp:
 "2018-02-01T17:21:09.000+08:00"}
    ]) {
    locationReadings {
```

```
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}
```

1 つのミューテーションで 10 個のセンサー読み取り値を送信しました。これらは 2 つのテーブルに分けられています。DynamoDB コンソールを使用して、locationReadings と temperatureReadings の両方のテーブルにデータが表示されることを確認します。

BatchDeleteItem - センサーの読み取り値の削除

同様に、センサーの読み取り値を一括して削除する必要もあります。これを行うために、Mutation.deleteReadings GraphQL フィールドを使用してみましょう。[Mutation.recordReadings] フィールドの横にある [アタッチ] を選択します。次の画面で、チュートリアルの最初で作成したのと同じ BatchTutorial データソースを選択します。

次のリクエストマッピングテンプレートを使用しましょう。

リクエストマッピングテンプレート

```
## Convert tempReadings arguments to DynamoDB primary keys
#set($tempReadings = [])
#foreach($reading in ${ctx.args.tempReadings})
    #set($pkey = {})
    $util.qr($pkey.put("sensorId", $reading.sensorId))
    $util.qr($pkey.put("timestamp", $reading.timestamp))
    $util.qr($tempReadings.add($util.dynamodb.toMapValues($pkey)))
#end

## Convert locReadings arguments to DynamoDB primary keys
#set($locReadings = [])
#foreach($reading in ${ctx.args.locReadings})
    #set($pkey = {})
    $util.qr($pkey.put("sensorId", $reading.sensorId))
```

```
    $util.qr($pkey.put("timestamp", $reading.timestamp))
    $util.qr($locReadings.add($util.dynamodb.toMapValues($pkey)))
#end

{
    "version" : "2018-05-29",
    "operation" : "BatchDeleteItem",
    "tables" : {
        "locationReadings": $utils.toJson($locReadings),
        "temperatureReadings": $utils.toJson($tempReadings)
    }
}
```

レスポンスマッピングテンプレートは Mutation.recordReadings に使用したものと同じです。

レスポンスマッピングテンプレート

```
## If there was an error with the invocation
## there might have been partial results
#if($ctx.error)
    ## Append a GraphQL error for that field in the GraphQL response
    $utils.appendError($ctx.error.message, $ctx.error.message)
#end
## Also return data for the field in the GraphQL response
$utils.toJson($ctx.result.data)
```

リゾルバーを保存し、 AWS AppSync コンソールのクエリページに移動します。では、いくつかのセンサーの読み取り値を削除しましょう!

次のミューテーションを実行します。

```
mutation deleteReadings {
  # Let's delete the first two readings we recorded
  deleteReadings(
    tempReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]
    locReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
```

```
      sensorId
      timestamp
      value
    }
  }
}
```

DynamoDB コンソールを通じて locationReadings と temperatureReadings テーブルから削除されたこれら 2 つの読み取り値を検証します。

BatchGetItem - 読み取り値の取得

Pet Health アプリのもう 1 つの一般的な処理として、特定の時刻にセンサーの読み取り値を取得します。スキーマの `Query.getReadings` GraphQL フィールドにリゾルバーをアタッチしましょう。[Attach (アタッチ)] を選択し、次の画面で、このチュートリアルの最初で作成したのと同じ `BatchTutorial` データソースを選択します。

次のリクエストマッピングテンプレートを追加しましょう。

リクエストマッピングテンプレート

```
## Build a single DynamoDB primary key,
## as both locationReadings and tempReadings tables
## share the same primary key structure
#set($pkey = {})
$util.qr($pkey.put("sensorId", $ctx.args.sensorId))
$util.qr($pkey.put("timestamp", $ctx.args.timestamp))

{
    "version" : "2018-05-29",
    "operation" : "BatchGetItem",
    "tables" : {
        "locationReadings": {
            "keys": [$util.dynamodb.toMapValuesJson($pkey)],
            "consistentRead": true
        },
        "temperatureReadings": {
            "keys": [$util.dynamodb.toMapValuesJson($pkey)],
            "consistentRead": true
        }
    }
}
```

BatchGetItem 処理を現在使用していることに注意してください。

SensorReading リストが返されることを選択しているため、レスポンスマッピングテンプレートには多少の相違が発生します。呼び出しの結果を必要な形式にマッピングしましょう。

レスポンスマッピングテンプレート

```
## Merge locationReadings and temperatureReadings
## into a single list
## __typename needed as schema uses an interface
#set($sensorReadings = [])

#foreach($locReading in $ctx.result.data.locationReadings)
    $util.qr($locReading.put("__typename", "LocationReading"))
    $util.qr($sensorReadings.add($locReading))
#end

#foreach($tempReading in $ctx.result.data.temperatureReadings)
    $util.qr($tempReading.put("__typename", "TemperatureReading"))
    $util.qr($sensorReadings.add($tempReading))
#end

$util.toJson($sensorReadings)
```

リゾルバーを保存し、 AWS AppSync コンソールのクエリページに移動します。では、センサーの読み取り値を取得しましょう!

次のクエリを実行します。

```
query getReadingsForSensorAndTime {
  # Let's retrieve the very first two readings
  getReadings(sensorId: 1, timestamp: "2018-02-01T17:21:06.000+08:00") {
    sensorId
    timestamp
    ...on TemperatureReading {
      value
    }
    ...on LocationReading {
      lat
      long
    }
  }
```

```
}
```

以上で、 AWS AppSync を使用した DynamoDB のバッチ処理のデモンストレーションが完了しました。

エラー処理

In AWS AppSync では、データソースオペレーションが部分的な結果を返すことがあります。部分的な結果という用語は、処理の出力がいくつかのデータと 1 つのエラーで構成されていることを表す場合に使用します。エラー処理はアプリケーションで異なるため、 AWS AppSync は、エラーを処理する手段をレスポンスマッピングテンプレート内に用意しています。リゾルバーの呼び出しエラーがある場合、これは context から `$ctx.error` として読み出せます。呼び出しエラーには必ずメッセージと型が含まれています。これらは `$ctx.error.message` と `$ctx.error.type` というプロパティとしてアクセスできます。レスポンスマッピングテンプレートの呼び出し中に、以下の 3 つの方法で部分的な結果を処理することができます。

1. データを返すだけで、呼び出しエラーは通知しない
2. レスポンスマッピングテンプレートの評価を停止することでエラーを発生させる (`$util.error(...)` を使用)。データは返さない。
3. エラーを付加し (`$util.appendError(...)` を使用)、データも返す

DynamoDB のバッチ処理を使用して、上記の 3 つの方法をそれぞれ試してみましょう!

DynamoDB のバッチ処理

DynamoDB のバッチ処理を使用すると、バッチを部分的に完了させることができます。つまり、リクエストされた項目またはキーの一部を未処理のままにすることができます。 AWS AppSync がバッチを完了できない場合、未処理の項目と呼び出しエラーがコンテキストに設定されます。

このチュートリアルの以前のセクションの `Query.getReadings` 処理で使用した `BatchGetItem` フィールドの設定を使用してエラー処理を実装します。ここでは、`Query.getReadings` フィールドの実行中に、`temperatureReadings` DynamoDB テーブルがプロビジョニングされたスループットを使い果たしたとします。DynamoDB は、 AWS AppSync がバッチ内の残りの要素を処理する 2 回目の試行で ProvisionedThroughputExceededException を発生させました。

次の JSON は、DynamoDB のバッチ処理の呼び出し後、レスポンスマッピングテンプレートの評価前にシリアル化された context を表しています。

```
{
```

```
  "arguments": {
    "sensorId": "1",
    "timestamp": "2018-02-01T17:21:05.000+08:00"
  },
  "source": null,
  "result": {
    "data": {
      "temperatureReadings": [
        null
      ],
      "locationReadings": [
        {
          "lat": 47.615063,
          "long": -122.333551,
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ]
    },
    "unprocessedKeys": {
      "temperatureReadings": [
        {
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ],
      "locationReadings": []
    }
  },
  "error": {
    "type": "DynamoDB:ProvisionedThroughputExceededException",
    "message": "You exceeded your maximum allowed provisioned throughput for a table or
 for one or more global secondary indexes. (...)"
  },
  "outErrors": []
}
```

context に関する注意事項がいくつかあります。

- by AWS AppSync `$ctx.error` のコンテキストで呼び出しエラーが設定され、エラータイプが DynamoDB:ProvisionedThroughputExceededException に設定されています。
- エラーが存在していても、結果はテーブルごとに `$ctx.result.data` にマッピングされます。

- 未処理のキーは $ctx.result.data.unprocessedKeys でアクセス可能です。ここでは、テーブルスループットが不十分なため、 AWS AppSync はキー (sensorId:1、timestamp:2018-02-01T17:21:05.000+08:00) を持つ項目を取得できませんでした。

注意 : BatchPutItem の場合、これは $ctx.result.data.unprocessedItems です。BatchDeleteItem の場合、これは $ctx.result.data.unprocessedKeys です。

3 つの異なる方法でこのエラーを処理しましょう。

1. 呼び出しエラーを通知しない

呼び出しエラーを処理せずにデータを返して、エラーを実質的に通知しません。指定した GraphQL フィールドの結果は常に成功とします。

記述するレスポンスマッピングテンプレートは通常どおりであり、結果のデータのみを扱います。

レスポンスマッピングテンプレート

```
$util.toJson($ctx.result.data)
```

GraphQL レスポンス

```
{
  "data": {
    "getReadings": [
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "lat": 47.615063,
        "long": -122.333551
      },
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  }
}
```

データのみが有効なため、エラーレスポンスにエラーは追加されません。

2. エラーを発生させ、テンプレートの実行を中断する

クライアントから見て、部分的な障害を完全な障害として扱う必要がある場合、テンプレートの実行を中断することでデータの返送を抑制することができます。この動作には、`$util.error(...)` ユーティリティメソッドを使用するのが最適です。

レスポンスマッピングテンプレート

```
## there was an error let's mark the entire field
## as failed and do not return any data back in the response
#if ($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type, null,
 $ctx.result.data.unprocessedKeys)
#end

$util.toJson($ctx.result.data)
```

GraphQL レスポンス

```
{
  "data": {
    "getReadings": null
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
```

```
        }
      ],
      "message": "You exceeded your maximum allowed provisioned throughput for a table
 or for one or more global secondary indexes. (...)"
    }
  ]
}
```

一部の結果が DynamoDB のバッチ処理から返されてもエラーが発生するように選択して、getReadings GraphQL フィールドが null になり、かつ GraphQL レスポンスの errors ブロックにエラーが追加されるようにします。

3. エラーを付加してデータとエラーの両方を返す

場合によっては、より優れたユーザーエクスペリエンスを提供するために、アプリケーションから部分的な結果を返すとともに、クライアントに未処理の項目を通知することができます。クライアントでは、再試行を実装することも、エラーを変換してエンドユーザーに通知することもできます。$util.appendError(...) はこの動作を可能とするユーティリティメソッドであり、アプリケーションの設計者は、テンプレートの評価を妨げることなく、context にエラーを付加できます。テンプレートを評価した後、 AWS AppSync はコンテキストエラーを GraphQL レスポンスのエラーブロックに追加して処理します。

レスポンスマッピングテンプレート

```
#if ($ctx.error)
    ## pass the unprocessed keys back to the caller via the `errorInfo` field
    $util.appendError($ctx.error.message, $ctx.error.type, null,
 $ctx.result.data.unprocessedKeys)
#end

$util.toJson($ctx.result.data)
```

ここでは、GraphQL レスポンスのエラーブロック内の呼び出しエラーと unprocessedKeys 要素の両方が転送されています。以下のレスポンスで分かるとおり、getReadings フィールドもまた locationReadings テーブルから部分的なデータを返します。

GraphQL レスポンス

```
{
  "data": {
    "getReadings": [
      null,
```

```
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
        }
      ],
      "message": "You exceeded your maximum allowed provisioned throughput for a table
 or for one or more global secondary indexes. (...)"
    }
  ]
}
```

AWS AppSync での DynamoDB トランザクションの実行

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync は、1 つのリージョン内の 1 つ以上のテーブルで Amazon DynamoDB トランザクションオペレーションの使用をサポートしています。サポートされている処理は、`TransactGetItems` および `TransactWriteItems` です。 AWS AppSync でこれらの機能を使用すると、次のようなタスクを実行できます。

- 単一クエリでキーのリストを渡し、テーブルからの結果を返す
- 単一クエリで 1 つ以上のテーブルからレコードを読み取る
- トランザクション内のレコードを 1 つまたは複数のテーブルにオールオアナッシング方式で書き込みます
- いくつかの条件が満たされたときにトランザクションを実行します

アクセス許可

他のリゾルバーと同様に、 AWS AppSync でデータソースを作成し、ロールを作成するか、既存のロールを使用する必要があります。トランザクションオペレーションでは DynamoDB テーブルごとに異なるアクセス許可が必要であるため、読み取りまたは書き込みアクションのために設定されたアクセス許可がロールに必要です。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:us-east-1:accountId:table/TABLENAME",
                "arn:aws:dynamodb:region:accountId:table/TABLENAME/*"
            ]
        }
    ]
```

```
}
```

注: ロールは AWS AppSync のデータソースに関連付けられ、フィールドのリゾルバーはデータソースに対して呼び出されます。DynamoDB に対して取得するよう設定されたデータソースは、設定を単純にするために 1 つのテーブルのみ指定されます。したがって、単一のリゾルバーで複数のテーブルに対してトランザクションオペレーションを実行する場合、これはより高度なタスクであり、リゾルバーが利用するすべてのテーブルへのアクセスをデータソースのロールに許可する必要があります。これは、上記の IAM ポリシーの Resource フィールドで行われます。テーブルに対するトランザクション呼び出しの設定は、リゾルバーのテンプレートで行います。これについては以下で説明します。

データソース

分かりやすくするために、このチュートリアルではすべてのリゾルバーに同じデータソースを使用します。[Data sources (データソース)] タブで、新しい DynamoDB データソースを作成し、TransactTutorial という名前を付けます。テーブル名はトランザクションオペレーションのリクエストマッピングテンプレートで指定するので、何でも構いません。ここでは、テーブル名に `empty` を使用しています。

savingAccounts および checkingAccounts というテーブルが 2 つできます。両方ともパーティションキーとして `accountNumber` 、transactionHistory テーブル`transactionId`と一緒に持ちます。

このチュートリアルでは、次のインラインポリシーを含むロールが使用できます。`region` および `accountId` をお客様のリージョンとアカウント ID に置き換えます。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
```

```
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:us-east-1:accountId:table/savingAccounts",
                "arn:aws:dynamodb:region:accountId:table/savingAccounts/*",
                "arn:aws:dynamodb:us-east-1:accountId:table/checkingAccounts",
                "arn:aws:dynamodb:region:accountId:table/checkingAccounts/*",
                "arn:aws:dynamodb:us-east-1:accountId:table/transactionHistory",
                "arn:aws:dynamodb:region:accountId:table/transactionHistory/*"
            ]
        }
    ]
}
```

トランザクション

この例では、コンテキストは従来の銀行取引で、次の目的で TransactWriteItems を使用します。

- 普通預金から当座預金への振替
- 取引ごとの新しい取引レコードの生成

次に、TransactGetItems を使用して、普通預金と当座預金の詳細を取得します。

> **Warning**
>
> TransactWriteItems は、競合の検出と解決とともに使用する場合はサポートされていません。これらの設定は、起こり得るエラーを防ぐために無効にする必要があります。

次のように GraphQL スキーマを定義します。

```
type SavingAccount {
    accountNumber: String!
    username: String
    balance: Float
}

type CheckingAccount {
    accountNumber: String!
    username: String
```

```
    balance: Float
}

type TransactionHistory {
    transactionId: ID!
    from: String
    to: String
    amount: Float
}

type TransactionResult {
    savingAccounts: [SavingAccount]
    checkingAccounts: [CheckingAccount]
    transactionHistory: [TransactionHistory]
}

input SavingAccountInput {
    accountNumber: String!
    username: String
    balance: Float
}

input CheckingAccountInput {
    accountNumber: String!
    username: String
    balance: Float
}

input TransactionInput {
    savingAccountNumber: String!
    checkingAccountNumber: String!
    amount: Float!
}

type Query {
    getAccounts(savingAccountNumbers: [String], checkingAccountNumbers: [String]):
 TransactionResult
}

type Mutation {
    populateAccounts(savingAccounts: [SavingAccountInput], checkingAccounts:
 [CheckingAccountInput]): TransactionResult
    transferMoney(transactions: [TransactionInput]): TransactionResult
}
```

```
schema {
    query: Query
    mutation: Mutation
}
```

TransactWriteItems - 口座の入力

口座間で振替を行うには、テーブルに詳細を入力する必要があります。これを実行するには、GraphQL オペレーション `Mutation.populateAccounts` を使用します。

[スキーマ] セクションで、`Mutation.populateAccounts` オペレーションの横にある [アタッチ] をクリックします。VTL ユニットリゾルバーに移動し、同じ `TransactTutorial` データソースを選択してください。

ここで、次のリクエストマッピングテンプレートを使用します。

リクエストマッピングテンプレート

```
#set($savingAccountTransactPutItems = [])
#set($index = 0)
#foreach($savingAccount in ${ctx.args.savingAccounts})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
 $util.dynamodb.toString($savingAccount.accountNumber)))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("username",
 $util.dynamodb.toString($savingAccount.username)))
    $util.qr($attributeValues.put("balance",
 $util.dynamodb.toNumber($savingAccount.balance)))
    #set($index = $index + 1)
    #set($savingAccountTransactPutItem = {"table": "savingAccounts",
        "operation": "PutItem",
        "key": $keyMap,
        "attributeValues": $attributeValues})
    $util.qr($savingAccountTransactPutItems.add($savingAccountTransactPutItem))
#end

#set($checkingAccountTransactPutItems = [])
#set($index = 0)
#foreach($checkingAccount in ${ctx.args.checkingAccounts})
    #set($keyMap = {})
```

```
    $util.qr($keyMap.put("accountNumber",
 $util.dynamodb.toString($checkingAccount.accountNumber)))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("username",
 $util.dynamodb.toString($checkingAccount.username)))
    $util.qr($attributeValues.put("balance",
 $util.dynamodb.toNumber($checkingAccount.balance)))
    #set($index = $index + 1)
    #set($checkingAccountTransactPutItem = {"table": "checkingAccounts",
        "operation": "PutItem",
        "key": $keyMap,
        "attributeValues": $attributeValues})
    $util.qr($checkingAccountTransactPutItems.add($checkingAccountTransactPutItem))
#end

#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountTransactPutItems))
$util.qr($transactItems.addAll($checkingAccountTransactPutItems))

{
    "version" : "2018-05-29",
    "operation" : "TransactWriteItems",
    "transactItems" : $util.toJson($transactItems)
}
```

さらに、以下のレスポンスマッピングテンプレートがあるとします。

レスポンスマッピングテンプレート

```
#if ($ctx.error)
    $util.appendError($ctx.error.message, $ctx.error.type, null,
 $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
    $util.qr($savingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($checkingAccounts = [])
#foreach($index in [3..5])
    $util.qr($checkingAccounts.add(${ctx.result.keys[$index]}))
#end
```

```
#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))

$util.toJson($transactionResult)
```

リゾルバーを保存し、 AWS AppSync コンソールのクエリセクションに移動して、アカウントを入力します。

次のミューテーションを実行します。

```
mutation populateAccounts {
  populateAccounts (
    savingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 100},
      {accountNumber: "2", username: "Amy", balance: 90},
      {accountNumber: "3", username: "Lily", balance: 80},
    ]
    checkingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 70},
      {accountNumber: "2", username: "Amy", balance: 60},
      {accountNumber: "3", username: "Lily", balance: 50},
    ]) {
    savingAccounts {
      accountNumber
    }
    checkingAccounts {
      accountNumber
    }
  }
}
```

1 つのミューテーションで 3 つの普通預金と 3 つの当座預金に入力しました。

DynamoDB コンソールを使用して、savingAccounts テーブルと checkingAccounts テーブルの両方にデータが表示されることを確認します。

TransactWriteItems - 振替

次のリクエストマッピングテンプレートを使用して `transferMoney` ミューテーションにリゾルバーをアタッチします。`amounts`、`savingAccountNumbers`、および `checkingAccountNumbers` の値は同じであることに注意してください。

```
#set($amounts = [])
#foreach($transaction in ${ctx.args.transactions})
    #set($attributeValueMap = {})
    $util.qr($attributeValueMap.put(":amount",
 $util.dynamodb.toNumber($transaction.amount)))
    $util.qr($amounts.add($attributeValueMap))
#end

#set($savingAccountTransactUpdateItems = [])
#set($index = 0)
#foreach($transaction in ${ctx.args.transactions})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
 $util.dynamodb.toString($transaction.savingAccountNumber)))
    #set($update = {})
    $util.qr($update.put("expression", "SET balance = balance - :amount"))
    $util.qr($update.put("expressionValues", $amounts[$index]))
    #set($index = $index + 1)
    #set($savingAccountTransactUpdateItem = {"table": "savingAccounts",
        "operation": "UpdateItem",
        "key": $keyMap,
        "update": $update})
    $util.qr($savingAccountTransactUpdateItems.add($savingAccountTransactUpdateItem))
#end

#set($checkingAccountTransactUpdateItems = [])
#set($index = 0)
#foreach($transaction in ${ctx.args.transactions})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
 $util.dynamodb.toString($transaction.checkingAccountNumber)))
    #set($update = {})
    $util.qr($update.put("expression", "SET balance = balance + :amount"))
    $util.qr($update.put("expressionValues", $amounts[$index]))
    #set($index = $index + 1)
    #set($checkingAccountTransactUpdateItem = {"table": "checkingAccounts",
        "operation": "UpdateItem",
        "key": $keyMap,
        "update": $update})

 $util.qr($checkingAccountTransactUpdateItems.add($checkingAccountTransactUpdateItem))
#end
```

```
#set($transactionHistoryTransactPutItems = [])
#foreach($transaction in ${ctx.args.transactions})
    #set($keyMap = {})
    $util.qr($keyMap.put("transactionId", $util.dynamodb.toString(${utils.autoId()})))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("from",
 $util.dynamodb.toString($transaction.savingAccountNumber)))
    $util.qr($attributeValues.put("to",
 $util.dynamodb.toString($transaction.checkingAccountNumber)))
    $util.qr($attributeValues.put("amount",
 $util.dynamodb.toNumber($transaction.amount)))
    #set($transactionHistoryTransactPutItem = {"table": "transactionHistory",
        "operation": "PutItem",
        "key": $keyMap,
        "attributeValues": $attributeValues})

 $util.qr($transactionHistoryTransactPutItems.add($transactionHistoryTransactPutItem))
#end

#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountTransactUpdateItems))
$util.qr($transactItems.addAll($checkingAccountTransactUpdateItems))
$util.qr($transactItems.addAll($transactionHistoryTransactPutItems))

{
    "version" : "2018-05-29",
    "operation" : "TransactWriteItems",
    "transactItems" : $util.toJson($transactItems)
}
```

単一の TransactWriteItems オペレーションで 3 つの銀行取引を行います。以下のレスポンスマッピングテンプレートを使用します。

```
#if ($ctx.error)
    $util.appendError($ctx.error.message, $ctx.error.type, null,
 $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
    $util.qr($savingAccounts.add(${ctx.result.keys[$index]}))
#end
```

```
#set($checkingAccounts = [])
#foreach($index in [3..5])
    $util.qr($checkingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($transactionHistory = [])
#foreach($index in [6..8])
    $util.qr($transactionHistory.add(${ctx.result.keys[$index]}))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))
$util.qr($transactionResult.put('transactionHistory', $transactionHistory))

$util.toJson($transactionResult)
```

次に AWS AppSync コンソールのクエリセクションに移動して、以下のように transferMoney ミューテーションを実行します。

```
mutation write {
  transferMoney(
    transactions: [
      {savingAccountNumber: "1", checkingAccountNumber: "1", amount: 7.5},
      {savingAccountNumber: "2", checkingAccountNumber: "2", amount: 6.0},
      {savingAccountNumber: "3", checkingAccountNumber: "3", amount: 3.3}
    ]) {
    savingAccounts {
      accountNumber
    }
    checkingAccounts {
      accountNumber
    }
    transactionHistory {
      transactionId
    }
  }
}
```

1 つのミューテーションで 2 つの銀行取引を送信しました。DynamoDB コンソールを使用して、savingAccounts、checkingAccounts、および transactionHistory の各テーブルにデータが表示されることを検証します。

TransactGetItems - 口座の取得

1 つの取引リクエストで普通預金口座と当座預金口座から詳細を取得するために、スキーマで Query.getAccounts GraphQL オペレーションにリゾルバーをアタッチします。[アタッチ] を選択し、次の画面で、このチュートリアルの最初で作成したのと同じ TransactTutorial データソースを選択します。テンプレートを次のように設定します。

リクエストマッピングテンプレート

```
#set($savingAccountsTransactGets = [])
#foreach($savingAccountNumber in ${ctx.args.savingAccountNumbers})
    #set($savingAccountKey = {})
    $util.qr($savingAccountKey.put("accountNumber",
 $util.dynamodb.toString($savingAccountNumber)))
    #set($savingAccountTransactGet = {"table": "savingAccounts", "key":
 $savingAccountKey})
    $util.qr($savingAccountsTransactGets.add($savingAccountTransactGet))
#end

#set($checkingAccountsTransactGets = [])
#foreach($checkingAccountNumber in ${ctx.args.checkingAccountNumbers})
    #set($checkingAccountKey = {})
    $util.qr($checkingAccountKey.put("accountNumber",
 $util.dynamodb.toString($checkingAccountNumber)))
    #set($checkingAccountTransactGet = {"table": "checkingAccounts", "key":
 $checkingAccountKey})
    $util.qr($checkingAccountsTransactGets.add($checkingAccountTransactGet))
#end

#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountsTransactGets))
$util.qr($transactItems.addAll($checkingAccountsTransactGets))

{
    "version" : "2018-05-29",
    "operation" : "TransactGetItems",
    "transactItems" : $util.toJson($transactItems)
}
```

レスポンスマッピングテンプレート

```
#if ($ctx.error)
```

```
    $util.appendError($ctx.error.message, $ctx.error.type, null,
 $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
    $util.qr($savingAccounts.add(${ctx.result.items[$index]}))
#end

#set($checkingAccounts = [])
#foreach($index in [3..4])
    $util.qr($checkingAccounts.add($ctx.result.items[$index]))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))

$util.toJson($transactionResult)
```

リゾルバーを保存し、 AWS AppSync コンソールのクエリセクションに移動します。普通預金口座と当座預金口座を取得するには、次のクエリを実行します。

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
      username
      balance
    }
  }
}
```

以上で、 AWS AppSync を使用した DynamoDB のトランザクションのデモンストレーションが完了しました。

AWS AppSync での HTTP リゾルバーの使用

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync を使用すると、サポートされているデータソース (Amazon DynamoDB AWS Lambda、Amazon OpenSearch Service、または Amazon Aurora) を使用して、GraphQL フィールドを解決するための任意の HTTP エンドポイントに加えて、さまざまなオペレーションを実行できます。HTTP エンドポイントが利用可能になったら、データソースを使用してこれに接続できます。その後、クエリ、ミューテーション、およびサブスクリプションなどの GraphQL オペレーションを実行するために、スキーマ内のリゾルバーを設定できます。このチュートリアルでは、いくつかの一般的な例を説明します。

このチュートリアルでは、 AWS AppSync GraphQL エンドポイントで REST API (Amazon API Gateway と Lambda により作成) を使用します。

ワンクリックでのセットアップ

(Amazon API Gateway と Lambda を使用して) HTTP エンドポイントを設定して AWS AppSync で GraphQL エンドポイントを自動的にセットアップする場合は、次の AWS CloudFormation テンプレートを使用できます。

REST API を作成する

次の AWS CloudFormation テンプレートを使用して、このチュートリアルで動作する REST エンドポイントを設定できます。

AWS CloudFormation スタックは次のステップを実行します。

1. マイクロサービス用のビジネスロジックを含む Lambda 関数を設定します。

2. 以下のエンドポイント/メソッド/コンテンツタイプを組み合わせて、API Gateway REST API をセットアップします。

API リソースパス	HTTP メソッド	サポートされているコンテンツタイプ
/v1/users	POST	application/json
/v1/users	GET	application/json
/v1/users/1	GET	application/json
/v1/users/1	PUT	application/json
/v1/users/1	DELETE	application/json

GraphQL API の作成

AWS AppSync で GraphQL API を作成するには：

- AWS AppSync コンソールを開き、API の作成を選択します。
- API 名として「UserData」と入力します。
- カスタムスキーマを選択します。
- [作成] を選択します。

AWS AppSync コンソールは、API キー認証モードを使用して新しい GraphQL API を作成します。このコンソールを使用して、残りの GraphQL API を設定し、このチュートリアルの残りの部分でクエリを実行できます。

GraphQL スキーマを作成する

GraphQL API を作成できたので、次に GraphQL スキーマを作成します。 AWS AppSync コンソールのスキーマエディタから、スキーマが次のスキーマと一致していることを確認します。

```
schema {
    query: Query
    mutation: Mutation
```

```
}

type Mutation {
    addUser(userInput: UserInput!): User
    deleteUser(id: ID!): User
}

type Query {
    getUser(id: ID): User
    listUser: [User!]!
}

type User {
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
}

input UserInput {
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
}
```

HTTP データソースを設定する

HTTP データソースを設定するには、以下の操作を行います。

- [DataSources (データソース)] タブで [New (新規)] を選択し、データソース名としてわかりやすい名前 (HTTP など) を入力します。
- [Data source type (データソースタイプ)] で「HTTP」を選択します。
- 作成された API Gateway エンドポイントに、エンドポイントを設定します。エンドポイントの一部にステージ名が含まれていないことを確認します。

注意: 現時点では、パブリックエンドポイントのみが AWS AppSync でサポートされています。

注： AWS AppSync サービスによって認識される認証機関の詳細については、「HTTPS エンドポイント AWS AppSync の によって認識される認証機関 (CA)」を参照してください。

リゾルバーの設定

このステップでは、http データソースを getUser クエリに接続します。

リゾルバーをセットアップするには、以下の手順に従います。

- [Schema (スキーマ)] タブを選択します。
- クエリタイプの右下のデータ型ペインで、getUserフィールドを見つけて、アタッチを選択します。
- [データソース名] で、[HTTP] を選択します。
- 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
    "version": "2018-05-29",
    "method": "GET",
    "params": {
        "headers": {
            "Content-Type": "application/json"
        }
    },
    "resourcePath": $util.toJson("/v1/users/${ctx.args.id}")
}
```

- 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
## return the body
#if($ctx.result.statusCode == 200)
    ##if response is 200
    $ctx.result.body
#else
    ##if response is not 200, append the response to error block.
    $utils.appendError($ctx.result.body, "$ctx.result.statusCode")
#end
```

- [Query (クエリ)] タブを選択して、以下のクエリを実行します。

```
query GetUser{
    getUser(id:1){
        id
        username
    }
}
```

これは以下のレスポンスを返します。

```
{
    "data": {
        "getUser": {
            "id": "1",
            "username": "nadia"
        }
    }
}
```

- [Schema (スキーマ)] タブを選択します。
- ミューテーションの右下のデータ型ペインで、AddUserフィールドを見つけて、アタッチを選択します。
- [データソース名] で、[HTTP] を選択します。
- 以下のコードを [リクエストマッピングテンプレートの設定] に貼り付けます。

```
{
    "version": "2018-05-29",
    "method": "POST",
    "resourcePath": "/v1/users",
    "params":{
      "headers":{
        "Content-Type": "application/json",
      },
      "body": $util.toJson($ctx.args.userInput)
    }
}
```

- 以下のコードを [レスポンスマッピングテンプレートの設定] に貼り付けます。

```
## Raise a GraphQL field error in case of a datasource invocation error
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
## if the response status code is not 200, then return an error. Else return the body
 **
#if($ctx.result.statusCode == 200)
    ## If response is 200, return the body.
    $ctx.result.body
#else
    ## If response is not 200, append the response to error block.
    $utils.appendError($ctx.result.body, "$ctx.result.statusCode")
#end
```

- [Query (クエリ)] タブを選択して、以下のクエリを実行します。

```
mutation addUser{
    addUser(userInput:{
        id:"2",
        username:"shaggy"
    }){
        id
        username
    }
}
```

これは以下のレスポンスを返します。

```
{
    "data": {
        "getUser": {
        "id": "2",
        "username": "shaggy"
        }
    }
}
```

AWS サービスの呼び出し

HTTP リゾルバーを使用して、 AWS サービスの GraphQL API インターフェイスを設定できます。への HTTP リクエストは、 AWS が送信したユーザーを識別できるように、署名バージョン 4 プロセスで署名 AWS する必要があります。 AWS AppSync は、IAM ロールを HTTP データソースに関連付けるときに、ユーザーに代わって署名を計算します。

HTTP リゾルバーを使用して AWS サービスを呼び出すには、次の 2 つの追加コンポーネントを提供します。

- AWS サービス APIs を呼び出すアクセス許可を持つ IAM ロール
- データソースの署名設定

たとえば、HTTP リゾルバーを使用して ListGraphqlApis オペレーションを呼び出す場合は、まず AWS AppSync が引き受ける IAM ロールを作成し、次のポリシーをアタッチします。

JSON

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

次に、 AWS AppSync の HTTP データソースを作成します。この例では、米国西部 (オレゴン) リージョンで AWS AppSync を呼び出します。http.json という名前のファイルで、署名リージョンとサービス名を含む以下の HTTP 設定を定義します。

```
{
    "endpoint": "https://appsync.us-west-2.amazonaws.com/",
    "authorizationConfig": {
```

```
        "authorizationType": "AWS_IAM",
        "awsIamConfig": {
            "signingRegion": "us-west-2",
            "signingServiceName": "appsync"
        }
    }
}
```

次に、 を使用して AWS CLI 、次のように関連付けられたロールを持つデータソースを作成します。

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

リゾルバーをスキーマの フィールドにアタッチする場合は、次のリクエストマッピングテンプレートを使用して AWS AppSync を呼び出します。

```
{
    "version": "2018-05-29",
    "method": "GET",
    "resourcePath": "/v1/apis"
}
```

このデータソースに対して GraphQL クエリを実行すると、 AWS AppSync は指定したロールを使用してリクエストに署名し、リクエストに署名を含めます。クエリは、その AWS リージョンのアカウントの AWS AppSync GraphQL APIsのリストを返します。

での Aurora Serverless v2 の使用 AWS AppSync

を使用して GraphQL API を Aurora Serverless データベースに接続します AWS AppSync。この統合により、GraphQL クエリ、ミューテーション、サブスクリプションを通じて SQL ステートメントを実行できます。これにより、リレーショナルデータを柔軟に操作できます。

> **Note**
>
> このチュートリアルでは、US-EAST-1 リージョンを使用しています。

利点

- GraphQL とリレーショナルデータベースのシームレスな統合
- GraphQL インターフェイスを使用して SQL オペレーションを実行する機能
- Aurora Serverless v2 によるサーバーレススケーラビリティ
- AWS Secrets Manager によるデータアクセスの保護
- 入力のサニタイズによる SQL インジェクションに対する保護
- フィルタリングや範囲操作などの柔軟なクエリ機能

一般的なユースケース

- リレーショナルデータ要件を使用したスケーラブルなアプリケーションの構築
- GraphQL の柔軟性と SQL データベース機能の両方を必要とする APIs の作成
- GraphQL ミューテーションとクエリによるデータオペレーションの管理
- 安全なデータベースアクセスパターンの実装

このチュートリアルでは、以下について説明します。

- Aurora Serverless v2 クラスターをセットアップする
- Data API 機能を有効にする
- データベース構造の作成と設定
- データベースオペレーションの GraphQL スキーマを定義する
- クエリとミューテーションのリゾルバーを実装する
- 適切な入力サニタイズによるデータアクセスの保護
- GraphQL インターフェイスを使用してさまざまなデータベースオペレーションを実行する

トピック

- データベースクラスターのセットアップ
- Data API を有効にする
- データベースとテーブルを作成する
- GraphQL スキーマ
- API をデータベースオペレーションに接続する

- API を使用してデータを変更する
- データの取得
- データアクセスを保護する

データベースクラスターのセットアップ

Amazon RDS データソースを に追加する前に AWS AppSync、まず Aurora Serverless v2 クラスターで Data API を有効にし、 を使用してシークレットを設定する必要がありますAWS Secrets Manager。Aurora Serverless v2 クラスターは、以下を使用して作成できます AWS CLI。

```
aws rds create-db-cluster \
    --db-cluster-identifier appsync-tutorial \
    --engine aurora-mysql \
    --engine-version 8.0 \
    --serverless-v2-scaling-configuration MinCapacity=0,MaxCapacity=1 \
    --master-username USERNAME \
    --master-user-password COMPLEX_PASSWORD \
    --enable-http-endpoint
```

これにより、クラスターの ARN が返されます。

クラスターを作成したら、次のコマンドを使用して Aurora Serverless v2 インスタンスを追加する必要があります。

```
aws rds create-db-instance \
    --db-cluster-identifier appsync-tutorial \
    --db-instance-identifier appsync-tutorial-instance-1 \
    --db-instance-class db.serverless \
    --engine aurora-mysql
```

> **Note**
>
> これらのエンドポイントのアクティブ化には時間がかかります。ステータスは、クラスターの接続とセキュリティタブの Amazon RDS コンソールで確認できます。次の AWS CLI コマンドを使用して、クラスターのステータスを確認することもできます。
>
> ```
> aws rds describe-db-clusters \
>     --db-cluster-identifier appsync-tutorial \
> ```

```
    --query "DBClusters[0].Status"
```

AWS Secrets Manager コンソールまたは AWS CLI を使用してシークレットを作成し、前のステップCOMPLEX_PASSWORDの USERNAMEと を使用して次のような入力ファイルを作成できます。

```
{
    "username": "USERNAME",
    "password": "COMPLEX_PASSWORD"
}
```

これをパラメータとして に渡します AWS CLI。

```
aws secretsmanager create-secret --name HttpRDSSecret --secret-string file://creds.json
 --region us-east-1
```

これにより、シークレットの ARN が返されます。

Aurora Serverless クラスターとシークレットの ARN をメモしておいてください。これらの ARN は後でデータソースを作成するときに AppSync コンソールで使用します。

Data API を有効にする

RDS のドキュメントの指示に従うことで、クラスターで Data API を有効にできます。Data API は AppSync データソースとして追加する前に有効にする必要があります。

データベースとテーブルを作成する

Data API を有効にしたら、 AWS CLIの `aws rds-data execute-statement` コマンドで使用できるようになります。これにより、Aurora Serverless クラスターが正しく設定されていることを AppSync API への追加前に確認できます。まず、`--sql` のようなパラメータで TESTDB というデータベースを作成します。

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789000:cluster:http-endpoint-test" \
--schema "mysql"  --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789000:secret:testHttp2-AmNvc1"  \
--region us-east-1 --sql "create DATABASE TESTDB"
```

これがエラーなしで実行されたら、create table コマンドを使用してテーブルを追加します。

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789000:cluster:http-endpoint-test" \
 --schema "mysql"  --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789000:secret:testHttp2-AmNvc1" \
 --region us-east-1 \
 --sql "create table Pets(id varchar(200), type varchar(200), price float)" --database
 "TESTDB"
```

すべてが問題なく実行されたら、AppSync API のデータソースとしてクラスターを追加する手順に進むことができます。

GraphQL スキーマ

Aurora Serverless Data API がテーブルで起動されて実行中になったところで、次は GraphQL スキーマを作成し、ミューテーションとサブスクリプションを実行するためのリゾルバーをアタッチします。 AWS AppSync コンソールで新しい API を作成し、スキーマページに移動して、次のように入力します。

```
type Mutation {
    createPet(input: CreatePetInput!): Pet
    updatePet(input: UpdatePetInput!): Pet
    deletePet(input: DeletePetInput!): Pet
}

input CreatePetInput {
    type: PetType
    price: Float!
}

input UpdatePetInput {
id: ID!
    type: PetType
    price: Float!
}

input DeletePetInput {
    id: ID!
}

type Pet {
    id: ID!
    type: PetType
```

```
    price: Float
}

enum PetType {
    dog
    cat
    fish
    bird
    gecko
}

type Query {
    getPet(id: ID!): Pet
    listPets: [Pet]
    listPetsByPriceRange(min: Float, max: Float): [Pet]
}

schema {
    query: Query
    mutation: Mutation
}
```

スキーマを保存し、[データソース] ページに移動して、新しいデータソースを作成します。データソースタイプとして [Relational database (リレーショナルデータベース)] を選択し、データソース名としてわかりやすい名前を入力します。前回の手順で作成したデータベースの名前と、そのデータベースを作成したクラスターのクラスター ARN を使用します。[Role (ロール)] では、AppSync により新しいロールを作成するか、以下のようなポリシーによりロールを作成できます。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "rds-data:DeleteItems",
                "rds-data:ExecuteSql",
                "rds-data:ExecuteStatement",
                "rds-data:GetItems",
                "rds-data:InsertItems",
                "rds-data:UpdateItems"
```

```
            ],
            "Resource": [
                "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster",
                "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster:*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": [
                "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret",
                "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret:*"
            ]
        }
    ]
}
```

このポリシーには、ロールにアクセス許可を付与する 2 つのステートメントがあります。最初のリソースは Aurora Serverless クラスターで、2 番目のリソースは AWS Secrets Manager ARN です。作成をクリックする前に AppSync データソース構成内の両方の ARN を指定する必要があります。

これをパラメータとして に渡します AWS CLI。

```
aws secretsmanager create-secret \
  --name HttpRDSSecret \
  --secret-string file://creds.json \
  --region us-east-1
```

これにより、シークレットの ARN が返されます。 AWS AppSync コンソールでデータソースを作成するときは、Aurora Serverless クラスターの ARN と のシークレットを書き留めます。

データベース構造を構築する

Data API を有効にしたら、 AWS CLIの `aws rds-data execute-statement` コマンドで使用できるようになります。これにより、API に追加する前に Aurora Serverless v2 クラスターが正しく設定されます AWS AppSync 。まず、次のように `--sql`パラメータを使用して TESTDB というデータベースを作成します。

```
aws rds-data execute-statement \
                --resource-arn "arn:aws:rds:us-east-1:111122223333:cluster:appsync-
tutorial" \
                --secret-arn "arn:aws:secretsmanager:us-
east-1:111122223333:secret:appsync-tutorial-rds-secret"  \
                --region us-east-1 \
                --sql "create DATABASE TESTDB"
```

これをエラーなしで実行する場合は、次の create table コマンドを使用してテーブルを追加します。

```
aws rds-data execute-statement \
      --resource-arn "arn:aws:rds:us-east-1:111122223333:cluster:http-endpoint-test" \
      --secret-arn "arn:aws:secretsmanager:us-east-1:111122223333:secret:testHttp2-
AmNvc1" \
      --region us-east-1 \
      --sql "create table Pets(id varchar(200), type varchar(200), price float)" \
      --database "TESTDB"
```

API インターフェイスの設計

Aurora Serverless v2 Data API がテーブルで起動して実行されたら、GraphQL スキーマを作成し、ミューテーションとサブスクリプションを実行するリゾルバーをアタッチします。 AWS AppSync コンソールで新しい API を作成し、コンソールのスキーマページに移動して、次のように入力します。

```
type Mutation {
        createPet(input: CreatePetInput!): Pet
        updatePet(input: UpdatePetInput!): Pet
        deletePet(input: DeletePetInput!): Pet
    }

    input CreatePetInput {
        type: PetType
        price: Float!
    }

    input UpdatePetInput {
        id: ID!
        type: PetType
        price: Float!
    }
```

```
input DeletePetInput {
    id: ID!
}

type Pet {
    id: ID!
    type: PetType
    price: Float
}

enum PetType {
    dog
    cat
    fish
    bird
    gecko
}

type Query {
    getPet(id: ID!): Pet
    listPets: [Pet]
    listPetsByPriceRange(min: Float, max: Float): [Pet]
}

schema {
    query: Query
    mutation: Mutation
}
```

スキーマを保存し、[データソース] ページに移動して、新しいデータソースを作成します。データソースタイプのリレーショナルデータベースを選択し、わかりやすい名前を指定します。前回の手順で作成したデータベースの名前と、そのデータベースを作成したクラスターのクラスター ARN を使用します。ロールでは、 で AWS AppSync 新しいロールを作成するか、次のようなポリシーで作成できます。

JSON

```
{
        "Version": "2012-10-17",
        "Statement": [
            {
```

```
                "Effect": "Allow",
                "Action": [
                    "rds-data:DeleteItems",
                    "rds-data:ExecuteSql",
                    "rds-data:ExecuteStatement",
                    "rds-data:GetItems",
                    "rds-data:InsertItems",
                    "rds-data:UpdateItems"
                ],
                "Resource": [
                    "arn:aws:rds:us-east-1:111122223333:cluster:mydbcluster",
                    "arn:aws:rds:us-east-1:111122223333:cluster:mydbcluster:*"
                ]
            },
            {
                "Effect": "Allow",
                "Action": [
                    "secretsmanager:GetSecretValue"
                ],
                "Resource": [
                    "arn:aws:secretsmanager:us-
east-1:111122223333:secret:mysecret",
                    "arn:aws:secretsmanager:us-
east-1:111122223333:secret:mysecret:*"
                ]
            }
        ]
    }
```

このポリシーには、ロールにアクセス許可を付与する 2 つのステートメントがあります。最初のリソースは Aurora Serverless v2 クラスターで、2 つ目は AWS Secrets Manager ARN です。Create をクリックする前に、 AWS AppSync データソース設定で両方の ARNs を指定する必要があります。

API をデータベースオペレーションに接続する

有効な GraphQL スキーマと RDS データソースができたので、リゾルバーをスキーマの GraphQL フィールドにアタッチできます。この API は以下の機能を提供します。

1. Mutation.createPet フィールドを使用してペットを作成する
2. Mutation.updatePet フィールドを使用してペットを更新する

3. Mutation.deletePet フィールドを使用してペットを削除する
4. Query.getPet フィールドを介して を使用して 1 つの を取得する
5. Query.listPets フィールドを使用してすべての を一覧表示する
6. Query.listPetsByPriceRange フィールドを使用して料金範囲内のペットを一覧表示する

Mutation.createPet

AWS AppSync コンソールのスキーマエディタから、右側の「 のリゾルバーのアタッチ」を選択しますcreatePet(input: CreatePetInput!): Pet。RDS データソースを選択します。[リクエストマッピングテンプレート] セクションで、以下のテンプレートを追加します。

```
#set($id=$utils.autoId())
{
"version": "2018-05-29",
    "statements": [
        "insert into Pets VALUES (:ID, :TYPE, :PRICE)",
        "select * from Pets WHERE id = :ID"
    ],
    "variableMap": {
        ":ID": "$ctx.args.input.id",
        ":TYPE": $util.toJson($ctx.args.input.type),
        ":PRICE": $util.toJson($ctx.args.input.price)
    }
}
```

システムは、ステートメント配列の順序に基づいて SQL ステートメントを順番に実行します。結果はその同じ順序で返されます。これはミューテーションであるため、挿入後に select ステートメントを実行し、GraphQL レスポンスマッピングテンプレートに入力するためにコミットされた値を取得します。

[レスポンスマッピングテンプレート] セクションで、以下のテンプレートを追加します。

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

ステートメントには 2 つの SQL クエリがあるため、データベースから返される行列の 2 番目の結果を $utils.rds.toJsonString($ctx.result))[1][0]) で指定する必要があります。

Mutation.updatePet

AWS AppSync コンソールのスキーマエディタから、 のリゾルバーをアタッチを選択しますupdatePet(input: UpdatePetInput!): Pet。RDS データソースを選択します。リクエストマッピングテンプレートセクションで、次のテンプレートを追加します。

```
{
"version": "2018-05-29",
    "statements": [
        $util.toJson("update Pets set type=:TYPE, price=:PRICE WHERE id=:ID"),
        $util.toJson("select * from Pets WHERE id = :ID")
    ],
    "variableMap": {
        ":ID": "$ctx.args.input.id",
        ":TYPE": $util.toJson($ctx.args.input.type),
        ":PRICE": $util.toJson($ctx.args.input.price)
    }
}
```

レスポンスマッピングテンプレートセクションで、次のテンプレートを追加します。

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

Mutation.deletePet

AWS AppSync コンソールのスキーマエディタから、 のリゾルバーをアタッチを選択しますdeletePet(input: DeletePetInput!): Pet。RDS データソースを選択します。リクエストマッピングテンプレートセクションで、次のテンプレートを追加します。

```
{
"version": "2018-05-29",
    "statements": [
        $util.toJson("select * from Pets WHERE id=:ID"),
        $util.toJson("delete from Pets WHERE id=:ID")
    ],
    "variableMap": {
        ":ID": "$ctx.args.input.id"
    }
}
```

レスポンスマッピングテンプレートセクションで、次のテンプレートを追加します。

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.getPet

スキーマのミューテーションが作成されたら、3 つのクエリを接続して、個々の項目、リストの取得方法、SQL フィルタリングの適用方法を示します。 AWS AppSync コンソールのスキーマエディタから、 のリゾルバーをアタッチを選択しますgetPet(id: ID!): Pet。RDS データソースを選択します。リクエストマッピングテンプレートセクションで、次のテンプレートを追加します。

```
{
"version": "2018-05-29",
        "statements": [
            $util.toJson("select * from Pets WHERE id=:ID")
    ],
    "variableMap": {
        ":ID": "$ctx.args.id"
    }
}
```

[レスポンスマッピングテンプレート] セクションで、以下のテンプレートを追加します。

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.listPets

AWS AppSync コンソールのスキーマエディタから、右側の「 のリゾルバーのアタッチ」を選択しますgetPet(id: ID!): Pet。RDS データソースを選択します。リクエストマッピングテンプレートセクションで、次のテンプレートを追加します。

```
{
    "version": "2018-05-29",
    "statements": [
        "select * from Pets"
    ]
}
```

レスポンスマッピングテンプレートセクションで、次のテンプレートを追加します。

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

Query.listPetsByPriceRange

AWS AppSync コンソールのスキーマエディタから、右側の「 のリゾルバーのアタッチ」を選択しますgetPet(id: ID!): Pet。RDS データソースを選択します。リクエストマッピングテンプレートセクションで、次のテンプレートを追加します。

```
{
    "version": "2018-05-29",
    "statements": [
            "select * from Pets where price > :MIN and price < :MAX"
    ],

    "variableMap": {
        ":MAX": $util.toJson($ctx.args.max),
        ":MIN": $util.toJson($ctx.args.min)
    }
}
```

[レスポンスマッピングテンプレート] セクションで、以下のテンプレートを追加します。

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

API を使用してデータを変更する

すべてのリゾルバーを SQL ステートメントで設定し、GraphQL API を Serverless Aurora Data API に接続したところで、ミューテーションとクエリの実行を開始できます。 AWS AppSync コンソールで、クエリタブを選択し、次のように入力して Pet を作成します。

```
mutation add {
    createPet(input : { type:fish, price:10.0 }){
        id
        type
        price
    }
}
```

レスポンスには、id、type、price が含まれています。

```
{
  "data": {
```

```
    "createPet": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "type": "fish",
      "price": "10.0"
    }
  }
}
```

この項目は updatePet ミューテーションを実行することで変更できます。

```
mutation update {
    updatePet(input : {
        id: ID_PLACEHOLDER,
        type:bird,
        price:50.0
    }){
        id
        type
        price
    }
}
```

事前のcreatePetオペレーションから返されたidを使用したことに注意してください。リゾルバーが `$util.autoId()` を利用したため、これがレコードに固有の値になります。同様の方法でレコードを削除できます。

```
mutation delete {
    deletePet(input : {id:ID_PLACEHOLDER}){
        id
        type
        price
    }
}
```

最初のミューテーションで price に異なる値を使用してレコードをいくつか作成したら、クエリをいくつか実行します。

データの取得

コンソールのクエリタブで、次のステートメントを使用して、作成したすべてのレコードを一覧表示します。

```
query allpets {
    listPets {
        id
        type
        price
    }
}
```

Query.listPetsByPriceRange のマッピングテンプレート`where price > :MIN and price < :MAX`に があった SQL WHERE 述語を、次の GraphQL クエリで活用します。

```
query petsByPriceRange {
    listPetsByPriceRange(min:1, max:11) {
        id
        type
        price
    }
}
```

price が 1 ドル以上、10 ドル未満のレコードのみが表示されます。最後に、以下のようにクエリを実行して個々のレコードを取得できます。

```
query onePet {
    getPet(id:ID_PLACEHOLDER){
        id
        type
        price
    }
}
```

データアクセスを保護する

SQL インジェクションは、データベースアプリケーションのセキュリティ上の脆弱性です。これは、攻撃者がユーザー入力フィールドを通じて悪意のある SQL コードを挿入した場合に発生します。これにより、データベースデータへの不正アクセスが可能になります。を使用して SQL インジェクション攻撃から保護する前に、すべてのユーザー入力`variableMap`を慎重に検証してサニタイズすることをお勧めします。変数マップが使用されていない場合は、GraphQL オペレーションの引数をサニタイズする責任があります。そのための 1 つの方法は、Data API に対して SQL ステートメントを実行する前に、リクエストマッピングテンプレートに入力固有の検証手順を提供することで

す。listPetsByPriceRange 例のリクエストマッピングテンプレートを変更する方法を見てみましょう。ユーザー入力だけに頼るのではなく、以下のことが可能です。

```
#set($validMaxPrice = $util.matches("\d{1,3}[,\\.]?(\\d{1,2})?",$ctx.args.maxPrice))

#set($validMinPrice = $util.matches("\d{1,3}[,\\.]?(\\d{1,2})?",$ctx.args.minPrice))


#if (!$validMaxPrice || !$validMinPrice)
    $util.error("Provided price input is not valid.")
#end
{
    "version": "2018-05-29",
    "statements": [
            "select * from Pets where price > :MIN and price < :MAX"
    ],

    "variableMap": {
        ":MAX": $util.toJson($ctx.args.maxPrice),
        ":MIN": $util.toJson($ctx.args.minPrice)
    }
}
```

Data API に対してリゾルバーを実行するときに不正な入力から保護するもう 1 つの方法は、プリペアードステートメントをストアドプロシージャおよびパラメータ化された入力と共に使用することです。例えば、listPets のリゾルバーで、select をプリペアードステートメントとして実行する以下のプロシージャを定義します。

```
CREATE PROCEDURE listPets (IN type_param VARCHAR(200))
  BEGIN
     PREPARE stmt FROM 'SELECT * FROM Pets where type=?';
     SET @type = type_param;
     EXECUTE stmt USING @type;
     DEALLOCATE PREPARE stmt;
  END
```

Aurora Serverless v2 インスタンスでこれを作成します。

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:xxxxxxxxxxxx:cluster:http-endpoint-test" \
--schema "mysql"  --secret-arn "arn:aws:secretsmanager:us-
east-1:xxxxxxxxxxxx:secret:httpendpoint-xxxxxx"  \
```

```
--region us-east-1  --database "DB_NAME" \
--sql "CREATE PROCEDURE listPets (IN type_param VARCHAR(200)) BEGIN PREPARE stmt FROM
 'SELECT * FROM Pets where type=?'; SET @type = type_param; EXECUTE stmt USING @type;
 DEALLOCATE PREPARE stmt; END"
```

その結果、listPets のリゾルバーコードは、ストアドプロシージャを呼び出すシンプルなものになりました。少なくとも、文字列入力では一重引用符をエスケープする必要があります。

```
#set ($validType = $util.isString($ctx.args.type) && !
$util.isNullOrBlank($ctx.args.type))
#if (!$validType)
    $util.error("Input for 'type' is not valid.", "ValidationError")
#end

{
    "version": "2018-05-29",
    "statements": [
        "CALL listPets(:type)"
    ]
    "variableMap": {
        ":type": $util.toJson($ctx.args.type.replace("'", "''"))
    }
}
```

エスケープ文字列の使用

一重引用符を使用して、 などの SQL ステートメントの文字列リテラルの開始と終了をマークします`'some string value'`。1 つ以上の一重引用符 (') を含む文字列値を文字列内で使用するには、それぞれを 2 つの一重引用符 ('') に置き換える必要があります。例えば、入力文字列が `Nadia's dog` の場合、SQL ステートメントでは次のようにエスケープします。

```
update Pets set type='Nadia''s dog' WHERE id='1'
```

AWS AppSync でのパイプラインリゾルバーの使用

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync は、ユニットリゾルバーを介して GraphQL フィールドを単一のデータソースにワイヤリングする簡単な方法を提供します。ただし、1 つのオペレーションを実行するだけでは不十分な場合があります。パイプラインリゾルバーは、データソースに対してオペレーションを順番に実行する機能を提供します。API で関数を作成し、パイプラインリゾルバーにアタッチします。各関数の実行結果は、実行する関数がなくなるまで、次の結果にパイプされます。パイプラインリゾルバーを使用すると、 AWS AppSync でより複雑なワークフローを直接構築できるようになりました。このチュートリアルでは、友人によって投稿された写真を投稿したり表示したりできる、シンプルな写真表示アプリケーションを作成します。

ワンクリックでのセットアップ

すべてのリゾルバーと必要な AWS リソースが設定された状態で AWS AppSync で GraphQL エンドポイントを自動的にセットアップする場合は、次の AWS CloudFormation テンプレートを使用できます。

このスタックはお客様のアカウントに以下のリソースを作成します。

- AWS AppSync がアカウントのリソースにアクセスするための IAM ロール
- 2 DynamoDB テーブル
- 1 つの Amazon Cognito ユーザープール
- 2 つの Amazon Cognito ユーザープールグループ
- 3 つの Amazon Cognito ユーザープールユーザー
- 1 AWS AppSync API

AWS CloudFormation スタック作成プロセスの最後に、作成された 3 人の Amazon Cognito ユーザーごとに 1 通の E メールが届きます。各 E メールには、 AWS AppSync コンソールへの Amazon Cognito ユーザーとしてログインするために使用する一時パスワードが含まれています。これらのパスワードを保存し、チュートリアルの残りで使用します。

手動セットアップ

AWS AppSync コンソールでstep-by-stepのプロセスを手動で実行する場合は、以下のセットアッププロセスに従います。

非 AWS AppSync リソースのセットアップ

API は、2 つの DynamoDB テーブル (写真を保存する pictures テーブルおよびユーザー間の関係を保存する friends テーブル) とやり取りします。API は、認証タイプとして Amazon Cognito ユーザープールを使用するように設定されています。次の AWS CloudFormation スタックは、 アカウントでこれらのリソースを設定します。

AWS CloudFormation スタック作成プロセスの最後に、作成された 3 人の Amazon Cognito ユーザーごとに 1 通の E メールが届きます。各 E メールには、 AWS AppSync コンソールへの Amazon Cognito ユーザーとしてログインするために使用する一時パスワードが含まれています。これらのパスワードを保存し、チュートリアルの残りで使用します。

GraphQL API の作成

AWS AppSync で GraphQL API を作成するには：

1. AWS AppSync コンソールを開き、Build From Scratch を選択し、Start を選択します。
2. API の名前を `AppSyncTutorial-PicturesViewer` に設定します。
3. [Create] (作成) を選択します。

AWS AppSync コンソールは、API キー認証モードを使用して新しい GraphQL API を作成します。このコンソールを使用して、残りの GraphQL API をセットアップでき、このチュートリアルの残りの部分でクエリを実行できます。

GraphQL API の設定

先ほど作成した Amazon Cognito ユーザープールで AWS AppSync API を設定する必要があります。

1. [Settings] (設定) タブを選択します。
2. [Authorization Type (承認タイプ)] セクションで、Amazon Cognito ユーザープールを選択します。
3. [ユーザープールの設定] で、AWS リージョンにUS-WEST-2を選択します。
4. AppSyncTutorial-UserPool ユーザープールを選択します。
5. デフォルトアクションとして拒否を選択します。
6. [AppId client regex (AppId クライアント正規表現)] フィールドは空白のままにします。
7. [Save] を選択します。

これで、承認タイプとして Amazon Cognito ユーザープールを使用するように API が設定されました。

DynamoDB テーブル用のデータソースの設定

DynamoDB テーブルを作成したら、コンソールで AWS AppSync GraphQL API に移動し、データソースタブを選択します。次に、先ほど作成した DynamoDB テーブルごとに、 AWS AppSync でデータソースを作成します。

1. [Data source (データソース)] タブを選択します。
2. [New (新規)] を選択して、新しいデータソースを作成します。
3. データソース名に、`PicturesDynamoDBTable` を入力します。
4. データソースのタイプとして [Amazon DynamoDB Table (Amazon DynamoDB テーブル)] を選択します。
5. リージョンとして [US-WEST-2 (米国西部 (オレゴン))] を選択します。
6. テーブルのリストから AppSyncTutorial-Pictures DynamoDB テーブルを選択します。
7. 「Create or use an existing role (作成または既存のロールの使用)」セクションで [Existing role (既存のロール)] を選択します。
8. CloudFormation テンプレートから先ほど作成したロールを選択します。ResourceNamePrefix を変更しなかった場合、ロールの名前は AppSyncTutorial-DynamoDBRole になっています。
9. [Create] (作成) を選択します。

friendsテーブルについても同じプロセスを繰り返します。CloudFormation スタックの作成時のResourceNamePrefixパラメータを変更しなかった場合、DynamoDB テーブルの名前はAppSyncTutorial-Friendsになります。

GraphQL スキーマの作成

データソースが DynamoDB テーブルに接続されたところで、GraphQL スキーマを作成しましょう。 AWS AppSync コンソールのスキーマエディタから、スキーマが次のスキーマと一致していることを確認します。

```
schema {
    query: Query
    mutation: Mutation
}
```

```
type Mutation {
    createPicture(input: CreatePictureInput!): Picture!
    @aws_auth(cognito_groups: ["Admins"])
    createFriendship(id: ID!, target: ID!): Boolean
    @aws_auth(cognito_groups: ["Admins"])
}

type Query {
    getPicturesByOwner(id: ID!): [Picture]
    @aws_auth(cognito_groups: ["Admins", "Viewers"])
}

type Picture {
    id: ID!
    owner: ID!
    src: String
}

input CreatePictureInput {
    owner: ID!
    src: String!
}
```

スキーマを保存するには、[Save Schema (スキーマの保存)] を選択します。

いくつかのスキーマフィールドには @aws_auth ディレクティブで注釈が付けられています。API のデフォルトのアクション設定は [拒否] に設定されているため、API は @aws_auth ディレクティブ内で指定されているグループのメンバーではないすべてのユーザーを拒否します。API の保護方法の詳細については、「セキュリティ」ページを参照してください。この場合、管理者ユーザーのみが Mutation.createPicture および Mutation.createFriendship フィールドにアクセスできますが、Admins または Viewers グループのメンバーであるユーザーは Query.getPicturesByOwner フィールドにアクセスできます。他のすべてのユーザーはアクセスできません。

リゾルバーの設定

有効な GraphQL スキーマと 2 つのデータソースを用意できたところで、スキーマでリゾルバーを GraphQL フィールドにアタッチできます。API は以下の機能を提供します。

- Mutation.createPicture フィールドで写真を作成する
- Mutation.createFriendship フィールドで友人関係を作成する
- Query.getPicture フィールドで写真を取得する

Mutation.createPicture

AWS AppSync コンソールのスキーマエディタから、右側の「 のリゾルバーのアタッチ」を選択しますcreatePicture(input: CreatePictureInput!): Picture!。DynamoDB PicturesDynamoDBTableデータソースを選択します。[リクエストマッピングテンプレート] セクションで、以下のテンプレートを追加します。

```
#set($id = $util.autoId())

{
    "version" : "2018-05-29",

    "operation" : "PutItem",

    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($id),
        "owner": $util.dynamodb.toDynamoDBJson($ctx.args.input.owner)
    },

    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args.input)
}
```

[レスポンスマッピングテンプレート] セクションで、以下のテンプレートを追加します。

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

写真の作成機能が実行されます。ランダムに生成された UUID を写真の ID として使用し、Cognito のユーザー名を写真の所有者として使用して、Pictures テーブルに写真を保存します。

Mutation.createFriendship

AWS AppSync コンソールのスキーマエディタから、右側の「 のリゾルバーのアタッチ」を選択しますcreateFriendship(id: ID!, target: ID!): Boolean。DynamoDB FriendsDynamoDBTableデータソースを選択します。[リクエストマッピングテンプレート] セクションで、以下のテンプレートを追加します。

```
#set($userToFriendFriendship = { "userId" : "$ctx.args.id", "friendId":
 "$ctx.args.target" })
```

```
#set($friendToUserFriendship = { "userId" : "$ctx.args.target", "friendId":
 "$ctx.args.id" })
#set($friendsItems = [$util.dynamodb.toMapValues($userToFriendFriendship),
 $util.dynamodb.toMapValues($friendToUserFriendship)])

{
    "version" : "2018-05-29",
    "operation" : "BatchPutItem",
    "tables" : {
        ## Replace 'AppSyncTutorial-' default below with the ResourceNamePrefix you
 provided in the CloudFormation template
        "AppSyncTutorial-Friends": $util.toJson($friendsItems)
    }
}
```

重要: BatchPutItem リクエストテンプレートには、DynamoDB テーブルの正確な名前が指定されている必要があります。デフォルトのテーブル名は AppSyncTutorial-Friends です。間違ったテーブル名を使用している場合、指定したロールを AppSync が引き受けようとすると エラーが発生します。

このチュートリアルでは、シンプルにすることを目的に、友人関係のリクエストが承認されたかのように処理を進め、関係のエントリを AppSyncTutorialFriends テーブルに直接保存します。

実際、友人関係は双方向であるため、関係ごとに 2 つの項目を保存します。多対多の関係を表すための Amazon DynamoDB のベストプラクティスの詳細については、「DynamoDB のベストプラクティス」を参照してください。

[レスポンスマッピングテンプレート] セクションで、以下のテンプレートを追加します。

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
true
```

注意: リクエストテンプレートに正しいテーブル名が含まれていることを確認してください。デフォルト名は AppSyncTutorial-Friends ですが、CloudFormation の ResourceNamePrefix パラメータを変更した場合、そのテーブル名が異なることがあります。

Query.getPicturesByOwner

友人関係と写真を用意できたところで、ユーザーが自分の友人の写真を表示できるようにする必要があります。この要件を満たすには、まずリクエスト者が所有者と友人であることを確認し、最後に写真のクエリを実行する必要があります。

この機能には 2 つのデータソースオペレーションが必要であるため、2 つの関数を作成します。最初の関数 isFriend は、リクエスト者と所有者が友人であるかどうかを確認します。2 番目の関数 getPicturesByOwner は、所有者 ID を指定してリクエストされた写真を取得します。Query.getPicturesByOwner フィールドで提案されたリゾルバーに対する以下の実行フローを見てみましょう。

1. Before マッピングテンプレート: コンテキストとフィールドの入力引数を準備します。
2. isFriend 関数: リクエスト者が写真の所有者かどうかを確認します。そうでない場合は、friends テーブルに対して DynamoDB GetItem オペレーションを実行して、リスクエスト者と所有者が友人であるかどうかを確認します。
3. GetPicturebyOwner 関数: owner-indexグローバルセカンダリインデックスの DynamoDB クエリ操作を使用して、ピクチャテーブルから画像を取得します。
4. After マッピングテンプレート: DynamoDB 属性が、想定される GraphQL タイプのフィールドに正しくマッピングされるように、写真の結果をマッピングします。

まず、関数を作成しましょう。

isFriend 関数

1. [関数] タブをクリックします。
2. [関数の作成] を選択して、関数を作成します。
3. データソース名に、`FriendsDynamoDBTable` を入力します。
4. 関数名として、「isFriend」と入力します。
5. リクエストマッピングテンプレートのテキスト領域内に、以下のテンプレートを貼り付けます。

```
#set($ownerId = $ctx.prev.result.owner)
#set($callerId = $ctx.prev.result.callerId)

## if the owner is the caller, no need to make the check
#if($ownerId == $callerId)
    #return($ctx.prev.result)
#end

{
    "version" : "2018-05-29",

    "operation" : "GetItem",
```

```
    "key" : {
        "userId" : $util.dynamodb.toDynamoDBJson($callerId),
        "friendId" : $util.dynamodb.toDynamoDBJson($ownerId)
    }
}
```

6. レスポンスマッピングテンプレートのテキスト領域内に、以下のテンプレートを貼り付けます。

```
#if($ctx.error)
    $util.error("Unable to retrieve friend mapping message: ${ctx.error.message}",
 $ctx.error.type)
#end

## if the users aren't friends
#if(!$ctx.result)
    $util.unauthorized()
#end

$util.toJson($ctx.prev.result)
```

7. [関数の作成] を選択します。

結果: isFriend 関数を作成しました。

getPicturesByOwner 関数

1. [関数] タブをクリックします。
2. [関数の作成] を選択して、関数を作成します。
3. データソース名に、PicturesDynamoDBTable を入力します。
4. 関数名として、「getPicturesByOwner」と入力します。
5. リクエストマッピングテンプレートのテキスト領域内に、以下のテンプレートを貼り付けます。

```
{
    "version" : "2018-05-29",

    "operation" : "Query",

    "query" : {
        "expression": "#owner = :owner",
        "expressionNames": {
            "#owner" : "owner"
```

```
        },
        "expressionValues" : {
            ":owner" : $util.dynamodb.toDynamoDBJson($ctx.prev.result.owner)
        }
    },

    "index": "owner-index"
}
```

6. レスポンスマッピングテンプレートのテキスト領域内に、以下のテンプレートを貼り付けます。

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end

$util.toJson($ctx.result)
```

7. [関数の作成] を選択します。

結果: getPicturesByOwner 関数を作成しました。関数が作成されたところで、パイプラインリゾルバーを Query.getPicturesByOwner フィールドにアタッチします。

AWS AppSync コンソールのスキーマエディタから、右側の「 のリゾルバーのアタッチ」を選択しますQuery.getPicturesByOwner(id: ID!): [Picture]。以下のページで、データソースのドロップダウンリストの下に表示される [Convert to pipeline resolver (パイプラインリゾルバーに変換)] リンクを選択します。Before マッピングテンプレートに、以下のものを使用します。

```
#set($result = { "owner": $ctx.args.id, "callerId": $ctx.identity.username })
$util.toJson($result)
```

After マッピングテンプレートのセクションに、以下のものを使用します。

```
#foreach($picture in $ctx.result.items)
    ## prepend "src://" to picture.src property
    #set($picture['src'] = "src://${picture['src']}")
#end
$util.toJson($ctx.result.items)
```

[Create Resolver (リゾルバー作成)] を選択します。最初のパイプラインリゾルバーが正常にアタッチされました。同じページで、前に作成した 2 つの関数を追加します。関数セクションで、[Add A Function (関数の追加)] を選択してから、最初の関数の名前として [isFriend] を選択または入力しま

す。getPicturesByOwner 関数のものと同じプロセスに従って、2 番目の関数を追加します。isFriend 関数がリストの最初に表示され、続いて getPicturesByOwner 関数が表示されていることを確認します。上下の矢印を使用して、パイプライン内の関数の実行順序に並べ替えることができます。

パイプラインリゾルバーが作成され、関数がアタッチされたところで、新しく作成した GraphQL API をテストしましょう。

GraphQL API をテストする

まず、作成した管理者ユーザーを使用していくつかのミューテーションを実行することで、写真と友人関係を入力する必要があります。 AWS AppSync コンソールの左側で、クエリタブを選択します。

createPicture ミューテーション

1. AWS AppSync コンソールで、クエリタブを選択します。
2. [Login With User Pools (ユーザープールでログイン)] を選択します。
3. モーダルで、CloudFormation スタックによって作成された Cognito Sample Client ID (37solo6mmhh7k4v63cqdfgdg5d など) を入力します。
4. CloudFormation スタックにパラメータとして渡したユーザー名を入力します。デフォルトは nadia です。
5. E メールに送信されて CloudFormation スタックへのパラメータとして渡した一時パスワード (UserPoolUserEmail など) を使用します。
6. [ログイン] を選択します。これで、ボタンの名前が Logout nadia に変更されているか、CloudFormation スタックの作成時に選択したユーザー名 (つまり UserPoolUsername) に変更されています。

pictures テーブルに入力するいくつかの createPicture ミューテーションを送信しましょう。コンソール内で以下の GraphQL クエリを実行します。

```
mutation {
  createPicture(input:{
    owner: "nadia"
    src: "nadia.jpg"
  }) {
    id
    owner
```

```
    src
  }
}
```

レスポンスは以下のようになります。

```
{
  "data": {
    "createPicture": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "owner": "nadia",
      "src": "nadia.jpg"
    }
  }
}
```

さらに写真をいくつか追加しましょう。

```
mutation {
  createPicture(input:{
    owner: "shaggy"
    src: "shaggy.jpg"
  }) {
    id
    owner
    src
  }
}
```

```
mutation {
  createPicture(input:{
    owner: "rex"
    src: "rex.jpg"
  }) {
    id
    owner
    src
  }
}
```

管理者ユーザーとして nadia を使用して 3 つの写真を追加しました。

createFriendship ミューテーション

友人関係エントリを追加しましょう。コンソールで以下のミューテーションを実行します。

注意: 管理者ユーザーとしてログインしている必要があります (デフォルトの管理者ユーザーは nadia です)。

```
mutation {
  createFriendship(id: "nadia", target: "shaggy")
}
```

レスポンスは以下のようになります。

```
{
  "data": {
    "createFriendship": true
  }
}
```

nadia と shaggy は友人です。rex はだれとも友人ではありません。

getPicturesByOwner クエリ

この手順では、Cognito ユーザープールと共に、このチュートリアルの始めに設定した認証情報を使用して、nadia ユーザーとしてログインします。nadia として、shaggy が所有する写真を取得します。

```
query {
    getPicturesByOwner(id: "shaggy") {
        id
        owner
        src
    }
}
```

nadia と shaggy は友人であるため、クエリからは対応する写真が返されます。

```
{
  "data": {
    "getPicturesByOwner": [
      {
```

```
        "id": "05a16fba-cc29-41ee-a8d5-4e791f4f1079",
        "owner": "shaggy",
        "src": "src://shaggy.jpg"
      }
    ]
  }
}
```

同様に、nadia が自分の写真を取得しようとした場合も、クエリは成功します。その場合、isFriend GetItem オペレーションを実行しないように、パイプラインリゾルバーは最適化されています。以下のクエリを試します。

```
query {
    getPicturesByOwner(id: "nadia") {
        id
        owner
        src
    }
}
```

API の ログ記録を有効にしている場合 ([Settings (設定)] ペインで)、デバッグレベルを [すべて] に設定し、同じクエリをもう一度実行すると、フィールド実行のログが返されます。ログを見ることで、リクエストマッピングテンプレートステージでisFriend関数が早期に返されたかどうか判断することができます。

```
{
  "errors": [],
  "mappingTemplateType": "Request Mapping",
  "path": "[getPicturesByOwner]",
  "resolverArn": "arn:aws:appsync:us-west-2:XXXX:apis/XXXX/types/Query/fields/
getPicturesByOwner",
  "functionArn": "arn:aws:appsync:us-west-2:XXXX:apis/XXXX/functions/
o2f42p2jrfdl3dw7s6xub2csdfs",
  "functionName": "isFriend",
  "earlyReturnedValue": {
    "owner": "nadia",
    "callerId": "nadia"
  },
  "context": {
    "arguments": {
      "id": "nadia"
    },
```

```
    "prev": {
      "result": {
        "owner": "nadia",
        "callerId": "nadia"
      }
    },
    "stash": {},
    "outErrors": []
  },
  "fieldInError": false
}
```

earlyReturnedValue キーは、#return ディレクティブによって返されたデータを表します。

最後に、rex は Viewers Cognito ユーザープールグループのメンバーです。rex はだれとも友人ではないため、shaggy または nadia によって所有されている写真にアクセスすることはできません。コンソールに rex としてログインし、以下のクエリを実行したとします。

```
query {
    getPicturesByOwner(id: "nadia") {
        id
        owner
        src
    }
}
```

以下の未承認エラーが発生します。

```
{
  "data": {
    "getPicturesByOwner": null
  },
  "errors": [
    {
      "path": [
        "getPicturesByOwner"
      ],
      "data": null,
      "errorType": "Unauthorized",
      "errorInfo": null,
      "locations": [
        {
```

```
          "line": 2,
          "column": 9,
          "sourceName": null
        }
      ],
      "message": "Not Authorized to access getPicturesByOwner on type Query"
    }
  ]
}
```

パイプラインリゾルバーを使用した複雑な承認が正常に実装されました。

のバージョニングされたデータソースでの Delta Sync オペレーションの使用 AWS AppSync

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync のクライアントアプリケーションは、GraphQL レスポンスをモバイル/ウェブアプリケーションのディスクにローカルにキャッシュすることでデータを保存します。バージョン管理されたデータソースと Sync オペレーションにより、お客様は単一のリゾルバーを使用して同期プロセスを実行することができます。これにより、クライアントは大量のレコードを含む可能性のある基本クエリでのローカルキャッシュをハイドレートし、最後のクエリ以降に変更されたデータのみを受信できます (差分更新)。クライアントが初期リクエストでキャッシュのベースハイドレートを実行し、別のリクエストでの増分更新を実行できるようにすることで、その計算をクライアントアプリケーションからバックエンドに移動できます。これは、頻繁にオンライン状態とオフラインの状態を切り替えるクライアントアプリケーションにとって効率が大きく向上します。

Delta Sync を実装するため、Sync クエリはバージョン管理されたデータソースで Sync オペレーションを使用します。 AWS AppSync ミューテーションがバージョニングされたデータソースの項目を変更すると、その変更の記録も Delta テーブルに保存されます。他のバージョニングされたデータソースには異なる Delta テーブル (タイプごとに 1 つ、ドメインエリアごとに 1 つなど) を使用するか、API には 1 つの Delta テーブルを使用するかを選択できます。 AWS AppSync では、プライマリキーの衝突を避けるために、複数の APIs に 1 つの Delta テーブルを使用することをお勧めします。

さらに、Delta Sync クライアントはサブスクリプションを引数として受け取ることもでき、オフラインからオンラインへの移行間でサブスクリプションの再接続と書き込みを調整します。そのために、Delta Sync はサブスクリプション (さまざまなネットワークエラーシナリオでの、エクスポネンシャルバックオフや、ジッターを伴う再試行など) を自動的に再開し、イベントをキューに保存します。その後、該当する差分クエリまたは基本クエリを実行してから、キュー内のイベントをマージし、最後に通常どおりにサブスクリプションを処理します。

Amplify DataStore などのクライアント設定オプションのドキュメントは、Amplify Amplify Frameworkのウェブサイトで入手できます。このドキュメントは、最適なデータアクセスのために Delta Sync クライアントと連携するように、バージョン管理された DynamoDB データソースと Sync オペレーションをセットアップする方法について概説しています。

ワンクリックでのセットアップ

すべてのリゾルバーと必要な AWS リソースが設定された状態で、 AWS AppSync で GraphQL エンドポイントを自動的にセットアップするには、次の AWS CloudFormation テンプレートを使用します。

このスタックはお客様のアカウントに以下のリソースを作成します。

- 2 つの DynamoDB テーブル (基本と差分)
- 1API キーを使用した AWS AppSync API
- DynamoDB テーブルのポリシーを割り当てた 1 つの IAM ロール

2 つのテーブルは、クライアントがオフラインのときに取得できなかったイベントのジャーナルとして機能する 2 番目のテーブルに、同期クエリを分割するために使用されます。差分テーブルに対するクエリが以降も効率的になるように、必要に応じてイベントの自動整理に Amazon DynamoDB TTL を使用します。TTL 時間は、データソースのニーズに合わせて設定できます (1 時間、1 日など)。

Schema

Delta Sync を実証するために、サンプルアプリケーションは DynamoDB の Base テーブルと Delta テーブルにバックアップされた Posts スキーマを作成します。 AWS AppSync はミューテーションを両方のテーブルに自動的に書き込みます。同期クエリは必要に応じて、ベーステーブルまたは差

分テーブルからレコードをプルします。また、1 つのサブスクリプションは、クライアントが再接続ロジックでこのテーブルをどのように活用できるかを定義します。

```
input CreatePostInput {
    author: String!
    title: String!
    content: String!
    url: String
    ups: Int
    downs: Int
    _version: Int
}

interface Connection {
  nextToken: String
  startedAt: AWSTimestamp!
}

type Mutation {
    createPost(input: CreatePostInput!): Post
    updatePost(input: UpdatePostInput!): Post
    deletePost(input: DeletePostInput!): Post
}

type Post {
    id: ID!
    author: String!
    title: String!
    content: String!
    url: AWSURL
    ups: Int
    downs: Int
    _version: Int
    _deleted: Boolean
    _lastChangedAt: AWSTimestamp!
}

type PostConnection implements Connection {
    items: [Post!]!
    nextToken: String
    startedAt: AWSTimestamp!
}
```

```
type Query {
    getPost(id: ID!): Post
    syncPosts(limit: Int, nextToken: String, lastSync: AWSTimestamp): PostConnection!
}

type Subscription {
    onCreatePost: Post
        @aws_subscribe(mutations: ["createPost"])
    onUpdatePost: Post
        @aws_subscribe(mutations: ["updatePost"])
    onDeletePost: Post
        @aws_subscribe(mutations: ["deletePost"])
}

input DeletePostInput {
    id: ID!
    _version: Int!
}

input UpdatePostInput {
    id: ID!
    author: String
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    _version: Int!
}

schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
}
```

GraphQL スキーマは標準的なものですが、先に進む前に注意することがいくつかあります。最初に、すべてのミューテーションは自動的にベーステーブルに書き込み、次に差分テーブルに書き込みます。基本テーブルは、実際の一元的なソースであり、一方、差分テーブルはジャーナルです。lastSync: AWSTimestamp を渡さない場合、 syncPosts クエリは基本テーブルに対して実行され、キャッシュがハイドレートされるだけでなく、グローバルキャッチアッププロセスとしても定期的に実行されます。これは、差分テーブルで設定された TTL 時間より長くクライアントが

オフラインになっているエッジケースに当てはまります。`lastSync: AWSTimestamp` を渡す場合、`syncPosts` クエリは差分テーブルに対して実行されます。クライアントはこのクエリを使用して、最後のオフライン以降に変更されたイベントを取得します。Amplify クライアントは自動的に `lastSync: AWSTimestamp` 値を渡し、適切にディスクに保持します。

POST 上の _deleted フィールドは、削除に使用します。クライアントがオフラインになり、レコードが基本テーブルから削除されると、この属性は、同期中のクライアントにローカルキャッシュから項目をエビクションするように通知します。クライアントが長期間オフラインであり、クライアントが Delta Sync クエリでこの値を取得する前に項目が削除された場合は、基本クエリのグローバルキャッチアップイベント (クライアントで設定可能) が実行され、キャッシュからその項目が削除されます。このフィールドがオプションとしてマークされているのは、同期クエリの実行時に、削除された項目がある場合にのみ値を返すためです。

ミューテーション

すべてのミューテーションについて、 AWS AppSync は基本テーブルで標準の Create/Update/Delete オペレーションを実行し、Delta テーブルに変更を自動的に記録します。データソースの `DeltaSyncTableTTL` 値を変更することで、レコードを保持する時間を短縮または延長できます。データの速度が速い組織では、この時間を短縮するのが妥当な場合があります。あるいは、クライアントが長期間オフラインになっている場合は、この時間を延長するのが賢明でしょう。

同期クエリ

基本クエリは、`lastSync` 値が指定されていない DynamoDB の同期オペレーションです。多くの組織にとって、基本クエリで十分です。基本クエリは起動時に実行され、以降は定期的に実行されるためです。

差分クエリは、`lastSync` 値が指定された DynamoDB の同期オペレーションです。差分クエリは、クライアントがオフライン状態からオンライン状態に戻るたびに実行されます (基本クエリが定期的にトリガーされていない場合)。クライアントは、データを同期するクエリを最後に正常に実行した時刻を自動的に追跡します。

差分クエリが実行されると、クエリのリゾルバーは ds_pk と ds_sk を使用して、クライアントが前回同期を実行してから変更されたレコードに対してのみクエリを実行します。クライアントは該当する GraphQL レスポンスを保存します。

同期クエリの実行の詳細については、同期オペレーションのドキュメントを参照してください。

例

まず、項目を作成するために createPost ミューテーションを呼び出すことから始めましょう。

```
mutation create {
  createPost(input: {author: "Nadia", title: "My First Post", content: "Hello World"})
 {
    id
    author
    title
    content
    _version
    _lastChangedAt
    _deleted
  }
}
```

このミューテーションの戻り値は次のようになります。

```
{
  "data": {
    "createPost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
      "author": "Nadia",
      "title": "My First Post",
      "content": "Hello World",
      "_version": 1,
      "_lastChangedAt": 1574469356331,
      "_deleted": null
    }
  }
}
```

ベーステーブルの内容を調べると、次のようなレコードがあります。

```
{
  "_lastChangedAt": {
    "N": "1574469356331"
  },
  "_version": {
    "N": "1"
  },
```

```
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "My First Post"
  }
}
```

差分テーブルの内容を調べると、次のようなレコードがあります。

```
{
  "_lastChangedAt": {
    "N": "1574469356331"
  },
  "_ttl": {
    "N": "1574472956"
  },
  "_version": {
    "N": "1"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "ds_pk": {
    "S": "AppSync-delta-sync-post:2019-11-23"
  },
  "ds_sk": {
    "S": "00:35:56.331:81d36bbb-1579-4efe-92b8-2e3f679f628b:1"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "My First Post"
```

```
  }
}
```

これで、クライアントが syncPosts クエリなどを実行してローカルデータストアをハイドレートするために実行する基本クエリをシミュレーションすることができます。

```
query baseQuery {
  syncPosts(limit: 100, lastSync: null, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
      _lastChangedAt
    }
    startedAt
    nextToken
  }
}
```

この基本クエリの戻り値は次のようになります。

```
{
  "data": {
    "syncPosts": {
      "items": [
        {
          "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
          "author": "Nadia",
          "title": "My First Post",
          "content": "Hello World",
          "_version": 1,
          "_lastChangedAt": 1574469356331
        }
      ],
      "startedAt": 1574469602238,
      "nextToken": null
    }
  }
}
```

差分クエリをシミュレーションするために後で startedAt 値を保存しますが、まずはテーブルを変更する必要があります。updatePost ミューテーションを使って既存の投稿を変更しましょう。

```
mutation updatePost {
  updatePost(input: {id: "81d36bbb-1579-4efe-92b8-2e3f679f628b", _version: 1, title:
 "Actually this is my Second Post"}) {
    id
    author
    title
    content
    _version
    _lastChangedAt
    _deleted
  }
}
```

このミューテーションの戻り値は次のようになります。

```
{
  "data": {
    "updatePost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
      "author": "Nadia",
      "title": "Actually this is my Second Post",
      "content": "Hello World",
      "_version": 2,
      "_lastChangedAt": 1574469851417,
      "_deleted": null
    }
  }
}
```

ここでベーステーブルの内容を調べると、更新された項目が表示されます。

```
{
  "_lastChangedAt": {
    "N": "1574469851417"
  },
  "_version": {
    "N": "2"
  },
  "author": {
```

```
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "Actually this is my Second Post"
  }
}
```

ここで差分テーブルの内容を調べると、次の 2 つのレコードがあります。

1. 項目が作成されたときのレコード
2. 項目が更新されたときのレコード

新しい項目は次のようになります。

```
{
  "_lastChangedAt": {
    "N": "1574469851417"
  },
  "_ttl": {
    "N": "1574473451"
  },
  "_version": {
    "N": "2"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "ds_pk": {
    "S": "AppSync-delta-sync-post:2019-11-23"
  },
  "ds_sk": {
    "S": "00:44:11.417:81d36bbb-1579-4efe-92b8-2e3f679f628b:2"
  },
```

```
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "Actually this is my Second Post"
  }
}
```

差分クエリをシミュレートして、クライアントがオフラインのときに発生した変更を取得できるようになりました。リクエストを行うために、基本クエリから返された値 `startedAt` を使用します。

```
query delta {
  syncPosts(limit: 100, lastSync: 1574469602238, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
    }
    startedAt
    nextToken
  }
}
```

この差分クエリの戻り値は次のようになります。

```
{
  "data": {
    "syncPosts": {
      "items": [
        {
          "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
          "author": "Nadia",
          "title": "Actually this is my Second Post",
          "content": "Hello World",
          "_version": 2
        }
      ],
      "startedAt": 1574470400808,
      "nextToken": null
    }
  }
```

```
}
```

の設定と設定 AWS AppSync

AWS AppSync では、次のことができます。

- 頻繁にリクエストされるが、リクエストによって変更される可能性は低いデータのキャッシュ。これにより、リゾルバーの負荷を軽減できます。詳細については、「the section called “サーバー側のキャッシュと API ペイロード圧縮の設定”」を参照してください。
- GraphQL オブジェクトをバージョン管理し、複数のクライアント間の競合を回避します。詳細については、「the section called “DynamoDB のバージョニング、競合検出、同期オペレーション”」を参照してください。
- カスタムドメイン名を使用して、GraphQL とリアルタイム API の両方で機能する覚えやすい単一のドメインを設定します。詳細については、「カスタムドメイン名を設定する」を参照してください。
- VPC を介した GraphQL API へのアクセスを許可します。詳細については、「AWS AppSync プライベート API の使用」を参照してください。
- との統合を通じて GraphQL APIs を共有します AWS Resource Access Manager。詳細については、AWS AppSync APIs」を参照してください。
- イントロスペクションを有効にし、クエリごとにクエリの深さとリゾルバーの制限を設定します。詳細については、「構成の制限」を参照してください。
- 環境変数を使用して、コードを更新せずに AWS AppSync リゾルバーと関数の動作を調整します。詳細については、「 での環境変数 AWS AppSyncの使用」を参照してください。

さらに、 には、ログ記録、モニタリング、トレース用の以下の標準 AWS ツール AWS AppSync が含まれています。

- ログイン AWS CloudTrail
- Amazon CloudWatch によるモニターリング
- を使用したトレース AWS X-Ray

AWS AppSync でのサーバー側のキャッシュと API ペイロード圧縮の設定

AWS AppSyncの AppSync のサーバー側のデータキャッシング機能は、データを高速インメモリキャッシュで利用できるようにし、パフォーマンスを高め、レイテンシーを減らします。これによ

り、データソースに直接アクセスする必要が少なくなります。キャッシングは、ユニットリゾルバーとパイプラインリゾルバーの両方で使用できます。

AWS AppSync では、API レスポンスを圧縮して、ペイロードコンテンツのロードとダウンロードを高速化することもできます。これにより、アプリケーションへの負担を軽減できると同時に、データ転送料金も削減できる可能性があります。圧縮動作は設定可能で、独自の判断で設定できます。

AWS AppSync API でのサーバー側のキャッシュと圧縮の望ましい動作を定義する方法については、このセクションを参照してください。

インスタンスのタイプ

AWS AppSync は AWS AppSync API と同じ AWS アカウントと AWS リージョンで Amazon ElastiCache (Redis OSS) インスタンスをホストします。

ElastiCache (Redis OSS) では、次のインスタンスタイプを使用できます。

small

1 vCPU、1.5 GiB RAM、低 ~ 中程度のネットワークパフォーマンス

medium

2 vCPU、3 GiB RAM、低 ~ 中程度のネットワークパフォーマンス

large

2 vCPU、12.3 GiB RAM、最大 10 ギガビットのネットワークパフォーマンス

xlarge

4 vCPU、25.05 GiB RAM、最大 10 ギガビットのネットワークパフォーマンス

2xlarge

8 vCPU、50.47 GiB RAM、最大 10 ギガビットのネットワークパフォーマンス

4xlarge

16 vCPU、101.38 GiB RAM、最大 10 ギガビットのネットワークパフォーマンス

8xlarge

32 vCPU、203.26 GiB RAM、10 ギガビットのネットワークパフォーマンス (一部のリージョンで使用できません)

12xlarge

48 vCPU、317.77 GiB RAM、10 ギガビットのネットワークパフォーマンス

> **Note**
>
> 従来は、特定のインスタンスタイプ (`t2.medium` など) を指定しました。2020 年 7 月時点で、これらのレガシーインスタンスタイプは引き続き利用可能になりますが、使用は推奨されなくなります。ここで説明するジェネリックインスタンスタイプを使用することをお勧めします。

キャッシュの動作

以下は、キャッシュに関連する動作です。

なし

サーバー側のキャッシュはありません。

完全なリクエストのキャッシュ

フルリクエストキャッシュは、リゾルバーの実行結果を個別にキャッシュするメカニズムです。この設定では、 はリクエスト中に呼び出されたすべてのリゾルバーの実行を AWS AppSync キャッシュし、各リゾルバーは個別にキャッシュされます。各リゾルバーのデータはデータソースから取得され、有効期限 (TTL) が切れるまでキャッシュに入力されます。後続の API リクエストの場合、特定のリゾルバーごとの結果がキャッシュから返されます。つまり、TTL の有効期限が切れていない限り、データソースに直接接続されません。 は の内容 AWS AppSync を使用し`context.arguments`、リゾルバーごとにキャッシュキーとして`context.identity`マッピングします。

リゾルバーごとのキャッシュ

この設定では、レスポンスをキャッシュするために、各リゾルバーを明示的にオプトインする必要があります。TTL とキャッシュキーはリゾルバーで指定できます。指定できるキャッシュキーは、これらのマップの最上位マップ `context.arguments`、`context.source`、`context.identity`、および/または文字列フィールドです。TTL 値は必須ですが、キャッシュキーはオプションです。キャッシュキーを何も指定しない場合、デフォルトは、`context.arguments`、`context.source`、`context.identity` マップの内容です。

たとえば、以下のようなコマンドを実行できます。

- context.arguments と context.source
- context.arguments と context.identity.sub
- context.arguments.id または context.arguments.InputType.id
- context.source.id と context.identity.sub
- context.identity.claims.username

TTL のみを指定し、キャッシュキーを指定しない場合、リゾルバーの動作はリクエスト全体をキャッシュする場合と同じになります。

オペレーションレベルのキャッシュ

オペレーションレベルのキャッシュは、GraphQL クエリオペレーションレスポンス全体を保存します。有効にすると、成功したクエリレスポンスは TTL の有効期限が切れるまでキャッシュされ、キャッシュ可能なレスポンスの最大サイズは 15 MB です。同じキャッシュキーを持つ後続のクエリリクエストの場合、TTL の有効期限が切れていない間、レスポンスはリゾルバーを実行せずにキャッシュから直接処理されます。

オペレーションレベルのキャッシュのキャッシュキーは、次の組み合わせを使用して生成されます。

- リクエストの JSON ペイロードからの特定の属性：
  - `query` 文字列
  - `operationName` 文字列
  - `variables` マップ
- `context.identity` マップ (IAM リクエストと Amazon Cognito リクエストの`context.identity.sourceIp`を除く)
- `context.request.headers` マップ (次のセクションにリストされている特定の予約済みヘッダーを除く)

リクエストで使用される認可タイプは、キャッシュキーにも影響します。IAM が認可したリクエストの場合、キャッシュキーには、許可されたリソースと拒否されたリソースのリストが追加されます。Lambda が認可したリクエストの場合、キャッシュキーには拒否されたフィールドのリストが追加されます。

キャッシュキーは、 で見つかったすべてのリクエストヘッダーを考慮します。ただし`context.request.headers`、通常、特定のリクエストに固有の以下の予約済みヘッダーは除きます。

- authorization
- cloudfront-forwarded-proto
- cloudfront-is-desktop-viewer
- cloudfront-is-mobile-viewer
- cloudfront-is-smarttv-viewer
- cloudfront-is-tablet-viewer
- cloudfront-viewer-asn
- cloudfront-viewer-country
- content-length
- host
- priority
- sec-ch-ua
- sec-ch-ua-mobile
- sec-ch-ua-platform
- via
- x-amz-cf-id
- x-amz-date
- x-amz-security-token
- x-amzn-appsync-is-vpce-request
- x-amzn-remote-ip
- x-amzn-requestid
- x-amzn-trace-id
- x-forwarded-for

キャッシュの存続時間 (TTL) 。

この設定は、キャッシュされたエントリがメモリに保存される時間を定義します。最大 TTL は 3,600 秒 (1 時間) です。その後、エントリは自動的に削除されます。

キャッシュ暗号化

AWS AppSyncのサーバー側のデータキャッシュ機能を使用する場合、保管時および転送時の暗号化は常に新しいキャッシュに対して有効になり、無効にすることはできません。

既存の API キャッシュで暗号化を有効にするには、キャッシュを削除してから再作成します。

キャッシュエントリを無効にするには、 AWS AppSync コンソールまたは AWS Command Line Interface () を使用してフラッシュキャッシュ API コールを実行できますAWS CLI。

詳細については、 AWS AppSync API リファレンス内の「ApiCache」データ型を参照してください。

キャッシュエビクション

AWS AppSync のサーバー側のキャッシュを設定するときに、最大 TTL を設定できます。これは、キャッシュされたエントリがメモリに保存される時間を定義します。キャッシュから特定のエントリを削除する必要がある場合は、リゾルバーのリクエストまたはレスポンスで AWS AppSync のevictFromApiCache拡張機能ユーティリティを使用できます。(たとえば、データソース内のデータが変更され、キャッシュエントリが古くなった場合など)。キャッシュからアイテムをエビクトするには、そのキーを知っている必要があります。このため、アイテムを動的にエビクトする必要がある場合は、リゾルバーごとのキャッシュを使用し、キャッシュにエントリを追加するために使用するキーを明示的に定義することをおすすめします。

キャッシュエントリーのエビクション

キャッシュからアイテムを削除するには、evictFromApiCache 拡張ユーティリティを使用します。タイプ名とフィールド名を指定し、キーと値の項目のオブジェクトを指定して、エビクトするエントリのキーを作成します。オブジェクト内の各キーは、キャッシュされたリゾルバーの cachingKey リストで使用されている context オブジェクトからの有効なエントリを表します。各値は、キーの値を構成するために使用される実際の値です。オブジェクト内の項目は、キャッシュされたリゾルバーの cachingKey リストにあるキャッシュキーと同じ順序で配置する必要があります。

例えば、以下のスキーマを参照してください。

```
type Note {
  id: ID!
  title: String
  content: String!
}

type Query {
  getNote(id: ID!): Note
}
```

```
type Mutation {
  updateNote(id: ID!, content: String!): Note
}
```

この例では、リゾルバーごとのキャッシュを有効にしてから、getNote クエリで有効にすることができます。次に、キャッシュキーを [context.arguments.id] で構成するように構成できます。

を取得しようとするとNote、 AWS AppSync はgetNoteクエリの id引数を使用してサーバー側のキャッシュでルックアップを実行します。

Note を更新するときは、次のリクエストでそのメモがバックエンドデータソースから取得されるように、特定のメモのエントリをエビクトする必要があります。これを行うには、リクエストハンドラーを作成する必要があります。

次の例は、このメソッドを使用してエビクションを処理する 1 つの方法を示しています。

```
import { util, Context } from '@aws-appsync/utils';
import { update } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 extensions.evictFromApiCache('Query', 'getNote', { 'ctx.args.id': ctx.args.id });
 return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });
}

export const response = (ctx) => ctx.result;
```

または、レスポンスハンドラーでエビクションを処理することもできます。

updateNote ミューテーションが処理されると、 AWS AppSync はエントリの削除を試みます。エントリが正常にクリアされると、レスポンスには削除されたエントリの数を示す apiCacheEntriesDeleted 値が extensions オブジェクトに含まれます。

```
"extensions": {  "apiCacheEntriesDeleted": 1}
```

ID に基づくキャッシュエントリのエビクション

context オブジェクトの複数の値に基づいてキャッシュキーを作成できます。

たとえば、Amazon Cognito ユーザープールをデフォルトの認証モードとして使用し、Amazon DynamoDB データソースを基盤とする次のスキーマを考えてみましょう。

```
type Note {
  id: ID! # a slug; e.g.: "my-first-note-on-graphql"
  title: String
  content: String!
}

type Query {
  getNote(id: ID!): Note
}

type Mutation {
  updateNote(id: ID!, content: String!): Note
}
```

Note オブジェクトタイプは DynamoDB テーブルに保存されます。このテーブルには、Amazon Cognito ユーザー名をプライマリキーとして使用し、Note の id (スラッグ) をパーティションキーとして使用する複合キーがあります。これはマルチテナントシステムで、複数のユーザーがプライベートオブジェクトをホストして更新できますが、プライベート Note オブジェクトは決して共有されません。

これは読み取り負荷の高いシステムであるため、getNote クエリはリゾルバーごとのキャッシュを使用してキャッシュされ、キャッシュキーは [context.identity.username, context.arguments.id] で構成されます。Note が更新されると、その特定のエントリを エビクトできます。Noteリゾルバーの cachingKeys リストで指定されているのと同じ順序でコンポーネントをオブジェクトに追加する必要があります。

次の例でこれを示します。

```
import { util, Context } from '@aws-appsync/utils';
import { update } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 extensions.evictFromApiCache('Query', 'getNote', {
  'ctx.identity.username': ctx.identity.username,
  'ctx.args.id': ctx.args.id,
 });
 return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });
}

export const response = (ctx) => ctx.result;
```

バックエンドシステムでも Note を更新してエントリをエビクトできます。例として、このミューテーションを取り上げます。

```
type Mutation {
  updateNoteFromBackend(id: ID!, content: String!, username: ID!): Note @aws_iam
}
```

エントリはエビクトできますが、キャッシュキーのコンポーネントは cachingKeys オブジェクトに追加できます。

次の例では、エビクションはリゾルバーの応答で行われます。

```
import { util, Context } from '@aws-appsync/utils';
import { update } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
    return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });
}

export function response(ctx) {
    extensions.evictFromApiCache('Query', 'getNote', {
        'ctx.identity.username': ctx.args.username,
        'ctx.args.id': ctx.args.id,
    });
    return ctx.result;
}
```

バックエンドデータが AWS AppSync の外部で更新された場合は、NONEデータソースを使用するミューテーションを呼び出すことで、キャッシュから項目を削除できます。

API レスポンスの圧縮

AWS AppSync を使用すると、クライアントは圧縮ペイロードをリクエストできます。要求された場合、API レスポンスは圧縮され、圧縮されたコンテンツが望ましいというリクエストに応答して返されます。圧縮された API レスポンスはより速く読み込まれ、コンテンツのダウンロードも速くなり、データ転送料金も抑えられる可能性があります。

> **Note**
>
> 圧縮は、2020 年 6 月 1 日以降に作成されたすべての新しい API で使用できます。

AWS AppSync はベストエフォートベースでオブジェクトを圧縮します。まれに、 AWS AppSync は現在の容量など、さまざまな要因に基づいて圧縮をスキップすることがあります。

AWS AppSync は、GraphQL クエリペイロードサイズを 1,000 ~ 10,000,000 バイトに圧縮できます。圧縮を有効にするには、クライアントは gzip 値を含む Accept-Encoding ヘッダーを送信する必要があります。圧縮は、応答 (gzip) Content-Encoding 内のヘッダーの値をチェックすることで確認できます。

AWS AppSync コンソールのクエリエクスプローラーは、デフォルトでリクエストのヘッダー値を自動的に設定します。応答が十分大きいクエリを実行した場合、ブラウザの開発者ツールを使用して圧縮を確認できます。

GraphQL およびリアルタイム API のカスタムドメイン名の設定

AWS AppSync を使用すると、カスタムドメイン名を使用して、GraphQL とリアルタイム APIs の両方で動作する単一の記憶に残るドメインを設定できます。

つまり、アカウントの AWS AppSync APIs に関連付けるカスタムドメイン名を作成することで、選択したドメイン名でシンプルで記憶に残るエンドポイント URLs を利用できます。

AWS AppSync API を設定すると、2 つのエンドポイントがプロビジョニングされます。

AWS AppSync GraphQL エンドポイント：

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync リアルタイムエンドポイント：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

カスタムドメイン名を使用すると、1 つのドメインを使用して両方のエンドポイントを操作できます。たとえば、api.example.com をカスタムドメインとして設定すると、次の URL を使用して GraphQL エンドポイントとリアルタイムエンドポイントの両方を操作できます。

AWS AppSync カスタムドメイン GraphQL エンドポイント：

```
https://api.example.com/graphql
```

AWS AppSync カスタムドメインのリアルタイムエンドポイント：

```
wss://api.example.com/graphql/realtime
```

> **Note**
>
> AWS AppSync APIsカスタムドメイン名の TLS 1.2 と TLS 1.3 のみをサポートします。

ドメイン名を登録および設定する

AWS AppSync APIs のカスタムドメイン名を設定するには、登録済みのインターネットドメイン名が必要です。 Amazon Route 53 domain registration を使用するか、お好みのサードパーティーのドメインレジストラを使用して、インターネットドメインを登録できます。Route 53 リゾルバーの詳細については、「Amazon Route 53 デベロッパーガイド」の「Amazon Route 53 とは？」を参照してください。

API のカスタムドメイン名は、登録されたインターネットドメインのサブドメイン名またはルートドメイン名 (「Zone Apex」など) にすることができます。 AWS AppSync でカスタムドメイン名を作成したら、DNS プロバイダーのリソースレコードを作成または更新して、API エンドポイントにマッピングする必要があります。このマッピングがない場合、カスタムドメイン名にバインドされた API リクエストは AWS AppSync に到達できません。

AWS AppSync でカスタムドメイン名を作成する

AWS AppSync API のカスタムドメイン名を作成すると、 Amazon CloudFront ディストリビューションが設定されます。カスタムドメイン名を CloudFront ディストリビューションドメイン名にマッピングするには、DNS レコードを設定する必要があります。このマッピングは、マッピングされた CloudFront ディストリビューションを介してカスタムドメイン名 AWS AppSync にバインドされた API リクエストをルーティングするために必要です。カスタムドメイン名の証明書を提供する必要もあります。

カスタムドメイン名を設定したり、証明書を更新したりするには、CloudFront ディストリビューションを更新し、使用する AWS Certificate Manager 予定の (ACM) 証明書を記述するアクセス許可が必要です。これらのアクセス許可を付与するには、次の AWS Identity and Access Management (IAM) ポリシーステートメントをアカウントの IAM ユーザー、グループ、またはロールにアタッチします。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowUpdateDistributionForAppSyncCustomDomainName",
            "Effect": "Allow",
            "Action": [
                "cloudfront:updateDistribution"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Sid": "AllowDescribeCertificateForAppSyncCustomDomainName",
            "Effect": "Allow",
            "Action": "acm:DescribeCertificate",
            "Resource": "arn:aws:acm:<region>:<account-id>:certificate/
<certificate-id>"
        }
    ]
}
```

AWS AppSync は、CloudFront ディストリビューションで Server Name Indication (SNI) を活用することで、カスタムドメイン名をサポートします。証明書の必須の形式や証明書の最大キー長など、CloudFront ディストリビューションでのカスタムドメイン名の使用の詳細については、「Amazon CloudFront デベロッパーガイド」の「CloudFront で HTTPS を使用する」を参照してください。

カスタムドメイン名を API のホスト名として設定するには、API 所有者は、カスタムドメイン名を対象とするus-east-1AWS リージョン (米国東部 (バージニア北部)) でパブリックまたはインポートされた ACM 証明書を提供する必要があります。ACM の詳細については、「AWS Certificate Manager ユーザーガイド」の「AWS Certificate Managerとは」を参照してください。

AWS AppSync でのワイルドカードカスタムドメイン名

AWS AppSync はワイルドカードカスタムドメイン名をサポートしています。ワイルドカードカスタムドメイン名を設定するには、カスタムドメインの最初のサブドメインとして、ワイ

ルドカード文字 (*) を指定します。これはルートドメインのすべての可能なサブドメインを表します。たとえば、ワイルドカードカスタムドメイン名として `*.example.com` を使用すると、`a.example.com`、`b.example.com`、`c.example.com` などのサブドメインが生成されます。これらのサブドメインはすべて同じドメインにルーティングされます。

AWS AppSync でワイルドカードカスタムドメイン名を使用するには、同じドメイン内の複数のサイトを保護できるワイルドカード名を含む ACM によって発行された証明書を提供する必要があります。詳細については、「AWS Certificate Manager ユーザーガイド」の「ACM 証明書の特徴」を参照してください。

の DynamoDB データソースのバージョニング、競合検出、同期オペレーション AWS AppSync

AWS AppSync の DynamoDB 用の高度なデータ管理機能は、バージョニングされたデータソース、競合の検出と解決、同期オペレーションの 3 つの主要な機能を活用します。これらのツールにより、分散環境でのデータ変更と同期を効率的に処理する堅牢でスケーラブルなアプリケーションが可能になります。

バージョン管理されたデータソースは、 AWS AppSync のデータ管理システムの基盤を形成します。この機能は、バージョニングメタデータを使用して DynamoDBアイテムを自動的に強化し、AWS AppSync ミューテーションによって行われた変更を Delta テーブルに記録し、削除された項目に対して「トゥームストーン」を維持します。デベロッパーは、削除された項目と変更ログの保持期間を設定し、データ整合性を確保しながらストレージを最適化できます。バージョン管理されたデータソースは、競合検出と同期オペレーションの実装を合理化し、高度なデータ処理のための強固な基盤を提供します。

競合の検出と解決のメカニズムは、同時書き込みが発生したときのデータ整合性を保護します。AWS AppSync には、オプティミスティック同時実行、自動マージ、Lambda ベースの解決の 3 つの戦略があります。オプティミスティック同時実行は競合するミューテーションを拒否するため、クライアントは更新されたデータを使用して再試行できます。Automerge は、データ型、マージリスト、セットユニオンの実行、既存のスカラー値の保存に基づいて、競合を自動的に解決します。Lambda ベースの解決により、複雑な競合シナリオのカスタムロジックが可能になります。これらのオプションにより、デベロッパーはデータの競合を柔軟に処理できるため、分散システム間の一貫性が確保されます。

同期オペレーションにより、クライアントアプリケーションの効率的なデータ取得と更新が可能になります。この機能を使用すると、クライアントは DynamoDB テーブルからすべての結果を取得し、

その後、最後のクエリ以降に変更されたデータのみを取得できます。 AWS AppSync は、提供された同期トークンに基づいて Base テーブルまたは Delta テーブルにアクセスするかどうかを決定し、パフォーマンスを最適化してデータ転送を削減します。

トピック

- での DynamoDB データソースのバージョニング AWS AppSync
- AWS AppSyncでの競合の検出と解決
- のバージョニングされたデータソースでの DynamoDB 同期オペレーションの使用 AWS AppSync

での DynamoDB データソースのバージョニング AWS AppSync

AWS AppSync は現在、DynamoDB データソースでのバージョニングをサポートしています。競合の検出、競合の解決、同期操作には、`Versioned` データソースが必要です。データソースでバージョニングを有効にすると、 AWS AppSync は自動的に以下を実行します。

- オブジェクトのバージョン管理メタデータを使用して項目を強化します。
- AWS AppSync ミューテーションを持つ項目に加えられた変更を Delta テーブルに記録します。
- 「廃棄」を使用して、ベーステーブル内の削除済み項目を、設定可能な期間保持します。

バージョン管理されたデータソースの設定

DynamoDB データソースでバージョン管理を有効にする場合は、次のフィールドを指定します。

`BaseTableTTL`

「廃棄」を使用して削除済み項目をベーステーブルに保持する分数。廃棄とは、項目が削除されたことを示すメタデータフィールドです。項目を削除したときにすぐに除外されるようにする場合は、この値を 0 に設定できます。このフィールドは必須です。

`DeltaSyncTableName`

AWS AppSync ミューテーションを持つ項目に加えられた変更が保存されるテーブルの名前。このフィールドは必須です。

`DeltaSyncTableTTL`

差分テーブルに項目を保持する分数。このフィールドは必須です。

Delta 同期テーブルのログ記録

AWS AppSync は現在、`PutItem`、、`UpdateItem`および DynamoDB オペレーションを使用したミューテーションの Delta Sync Logging `DeleteItem` をサポートしています。

AWS AppSync ミューテーションがバージョニングされたデータソースの項目を変更すると、その変更のレコードが差分更新用に最適化された Delta テーブルに保存されます。バージョン管理された他のデータソースに異なる Delta テーブル (タイプごとに 1 つ、ドメイン領域ごとに 1 つなど) を使用するか、API に 1 つの Delta テーブルを使用するかを選択できます。 AWS AppSync では、プライマリキーの衝突を避けるためAPIs に 1 つの Delta テーブルを使用することをお勧めします。

このテーブルに必要なスキーマは次のとおりです。

ds_pk

パーティションキーとして使用される文字列値。これは、Base データソース名と、変更が発生した日付の ISO 8601 形式を連結することによって作成されます (例: `Comments:2019-01-01`)。

VTL マッピングテンプレートの `customPartitionKey` フラグをパーティションキーが列名として設定されると (「AWS AppSync デベロッパーガイド」の「DynamoDB のリゾルバーのマッピングテンプレートリファレンス」を参照)、ds_pk の形式が変更され、文字列は、Base テーブルの新しいレコードのパーティションキーの値を追加して作成されます。たとえば、Base テーブルのレコードのパーティションキー値が 1aで、ソートキー値が 2b の場合、文字列の新しい値は `Comments:2019-01-01:1a` になります。

ds_sk

ソートキーとして使用される文字列値。これは、変更が発生した時刻の ISO 8601 形式、項目のプライマリキー、および項目のバージョンを連結することによって作成されます。これらのフィールドの組み合わせにより、Delta テーブルのすべてのエントリの一意性が保証されます (たとえば、時刻が `09:30:00`、ID が 1a、バージョン 2 の場合、`09:30:00:1a:2` になります)。

VTL マッピングテンプレートの `customPartitionKey` フラグをパーティションキーが列名として設定されると (「AWS AppSync デベロッパーガイド」の「DynamoDB のリゾルバーのマッピングテンプレートリファレンス」を参照)、ds_sk の形式が変更され、文字列は、コンビネーションキーの値を Base テーブルのソートキーの値に置き換えることで作成されます。上記の例を使用すると、Base テーブルのレコードのパーティションキー値が 1aで、ソートキー値が 2b の場合、文字列の新しい値は `09:30:00:2b:3` になります。

_ttl

差分テーブルから項目を削除する時刻のタイムスタンプをエポック秒単位で保存する数値。この値は、変更が発生したときに、データソースで設定された DeltaSyncTableTTL 値を加算することによって決定されます。このフィールドは DynamoDB TTL 属性として設定する必要があります。

ベーステーブルで使用するように設定された IAM ロールには、差分テーブルを操作するためのアクセス権限も含まれている必要があります。この例では、Comments という名前の基本テーブルと ChangeLog という名前の差分テーブルへの許可ポリシーが表示されます。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:000000000000:table/Comments",
                "arn:aws:dynamodb:us-east-1:000000000000:table/Comments/*",
                "arn:aws:dynamodb:us-east-1:000000000000:table/ChangeLog",
                "arn:aws:dynamodb:us-east-1:000000000000:table/ChangeLog/*"
            ]
        }
    ]
}
```

バージョン管理されたデータソースメタデータ

AWS AppSync は、ユーザーに代わってVersionedデータソースのメタデータフィールドを管理します。これらのフィールドを自分で変更すると、アプリケーションにエラーが発生したり、データが失われたりする可能性があります。これらのフィールドには、以下が含まれます。

_version

一定間隔で増加するカウンタ。項目に変更が発生するたびに更新されます。

_lastChangedAt

項目が最後に変更されたときのタイムスタンプをエポックミリ秒単位で保存する数値。

_deleted

項目が削除されたことを示すブール型の「廃棄」値。これは、削除された項目をローカルデータストアからエビクションするために、アプリケーションが使用できます。

_ttl

基になるデータソースから項目を削除する時刻のタイムスタンプをエポック秒単位で保存する数値。

ds_pk

差分テーブルのパーティションキーとして使用される文字列値。

ds_sk

差分テーブルのソートキーとして使用される文字列値。

gsi_ds_pk

グローバルセカンダリインデックスをパーティションキーとしてサポートするために生成される文字列値属性。これは、VTL マッピングテンプレートで `customPartitionKey` および `populateIndexFields` フラグの両方が有効になっている場合にのみ含まれます (「AWS AppSync デベロッパーガイド」の「DynamoDB のリゾルバーのマッピングテンプレートリファレンス」を参照)。有効にすると、Base データソース名と、変更が発生した日付の ISO 8601 形式を連結することによって値が作成されます (たとえば、Base テーブルの名前が Comments の場合、このレコードは `Comments:2019-01-01` として設定されます)。

gsi_ds_sk

グローバルセカンダリインデックスをソートキーとしてサポートするために生成される文字列値属性。これは、VTL マッピングテンプレートで `customPartitionKey` および

`populateIndexFields` フラグの両方が有効になっている場合にのみ含まれます (「AWS AppSync デベロッパーガイド」の「DynamoDB のリゾルバーのマッピングテンプレートリファレンス」を参照)。有効にすると、変更が発生した時刻の ISO 8601 形式、Base テーブル内の項目のパーティションキー、Base テーブル内の項目のソートキー、および項目のバージョンを連結することによって値が作成されます (たとえば、時刻が `09:30:00` で、パーティションキー値が `1a` で、ソートキーの値が `2b` で、バージョンが `3` の場合、`09:30:00:1a#2b:3` になります)。

これらのメタデータフィールドは、基盤となるデータソース内の項目の全体的なサイズに影響します。 AWS AppSync では、アプリケーションを設計するときに、バージョニングされたデータソースメタデータ用に 500 バイト + ストレージの最大プライマリキーサイズを予約することをお勧めします。このメタデータをクライアントアプリケーションで使用するには、GraphQL タイプとミューテーションの選択セットに、`_version`、`_lastChangedAt`、および `_deleted` フィールドを含めます。

AWS AppSyncでの競合の検出と解決

AWS AppSync で同時書き込みが発生した場合は、更新を適切に処理するように競合検出および競合解決戦略を設定できます。競合の検出では、ミューテーションがデータソースに実際に書き込まれた項目と競合しているかどうかが判断されます。競合の検出を有効にするには、`conflictDetection` フィールドの SyncConfig の値を `VERSION` に設定します。

競合の解決は、競合が検出された場合に実行されるアクションです。これは、SyncConfig の Conflict Handler フィールドを設定することによって決定されます。次の 3 つの競合の解決戦略があります。

- OPTIMISTIC_CONCURRENCY
- AUTOMERGE
- LAMBDA

バージョンは書き込みオペレーション中に AWS AppSync によって自動的に増分されるため、クライアントやバージョン対応データソースで設定されたリゾルバーの外部では変更しないでください。変更すると、システムの整合性動作が変更され、データが失われる可能性があります。

オプティミスティック同時実行

オプティミスティック同時実行は、 AWS AppSync がバージョニングされたデータソースに提供する競合解決戦略です。競合リゾルバーがオプティミスティック同時実行に設定されている場合、着信

ミューテーションのバージョンがオブジェクトの実際のバージョンとは異なることが検出されると、競合ハンドラーは着信リクエストを拒否します。GraphQL レスポンス内部では、最新バージョンを持つサーバー上の既存の項目が提供されます。クライアントは、この競合を口ーカルで処理し、項目の更新バージョンでミューテーションを再試行することが期待されます。

Automerge

Automerge により、開発者は他の方法で解決できなかった競合を手動でマージするクライアント側のロジックを記述しなくても、競合の解決戦略を簡単に設定することができます。Automerge は、データをマージして競合を解決するときに、厳密なルールセットに従います。Automerge の理念は、GraphQL フィールドの基になるデータ型を中心に展開されています。その内容は次のとおりです。

- スカラーフィールドでの競合: GraphQL スカラーまたはコレクションではないフィールド (リスト、セット、マップなど)。スカラーフィールドの入力値を拒否し、サーバーに存在する値を選択します。
- リストの競合: GraphQL 型とデータベース型はリストです。着信リストをサーバー上の既存のリストと連結します。着信ミューテーションのリスト値は、サーバー内のリストの最後に追加されます。重複した値は保持されます。
- セットの競合: GraphQL 型はリストであり、データベース型はセットです。着信セットとサーバー内の既存のセットを使用して、セットユニオンを適用します。これは、重複するエントリを持たないという、セットのプロパティに準拠しています。
- 着信ミューテーションが項目に新しいフィールドを追加するか、値が `null` のときにフィールドに対して着信ミューテーションが実行されたときは、それを既存の項目にマージします。
- マップ上の競合: データベースの基になるデータ型がマップ (キーと値のドキュメント) の場合、マップの各プロパティを解析して処理する上記のルールを適用します。

Automerge は、更新されたバージョンのリクエストを自動的に検出、マージ、再試行するように設計されており、クライアントは競合するデータを手動でマージする必要がありません。

Automerge がスカラー型で競合を処理する方法の例を示します。出発点として、次のレコードを使用します。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
```

```
  "_version" : 4
}
```

クライアントがまだサーバーと同期していないため、着信ミューテーションが項目を更新しようとしている可能性がありますが、それは古いバージョンです。以下のようになります。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 55,
  "_version" : 2
}
```

着信リクエストの古いバージョン 2 に注意してください。このフローの間、Automerge は jersey フィールドの 55 への更新を拒否し、値を 5 のままにすることでデータがマージされます。その結果、項目の次のイメージがサーバーに保存されます。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 5 # version is incremented every time automerge performs a merge that is
 stored on the server.
}
```

上記の項目の状態がバージョン 5 で、次のイメージで項目を変更しようとする着信ミューテーションがあるとします。

```
{
  "id" : 1,
  "name" : "Shaggy",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner"] # underlying data type is a Set
  "points": [24, 30, 27] # underlying data type is a List
  "_version" : 3
}
```

着信ミューテーションには、注目すべき 3 つのポイントがあります。スカラーという名前が変更されましたが、2 つの新しいフィールドである、セットの interests およびリストの points が追加されました。このシナリオでは、バージョンの不一致が原因で競合が検出されます。Automerge はその

プロパティに準拠し、名前の変更を拒否します。これは名前がスカラーで、競合しないフィールドのアドオンであるためです。これにより、サーバーに保存された項目は次のように表示されます。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner"] # underlying data type is a Set
  "points": [24, 30, 27] # underlying data type is a List
  "_version" : 6
}
```

項目の更新されたイメージはバージョン 6 となりました。ここで、着信ミューテーション (別のバージョンと不一致) が項目を次のように変換しようとするとします。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "brunch"] # underlying data type is a Set
  "points": [30, 35] # underlying data type is a List
  "_version" : 5
}
```

ここで interests の着信フィールドは、サーバーに存在する 1 つの重複値と 2 つの新しい値を持っていることがわかります。この場合、基になるデータ型はセットであるため、Automerge はサーバーに存在する値と着信リクエストの値を結合し、重複値を除外します。同様に、重複する値が 1 つと新しい値が 1 つある points フィールドには競合があります。しかし、ここで基になるデータ型はリストであるため、Automerge は着信リクエストのすべての値を、サーバーにすでに存在する値の最後に追加します。サーバーに保存されたマージ後のイメージは、次のようになります。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a
 Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "_version" : 7
}
```

次に、バージョン 8 で、サーバーに保存された項目が次のようになると仮定します。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a
 Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "stats": {
      "ppg": "35.4",
      "apg": "6.3"
  }
  "_version" : 8
}
```

しかし、着信リクエストは、次のイメージで項目を更新しようとしますが、再びバージョンの不一致が発生します。

```
{
  "id" : 1,
  "name" : "Nadia",
  "stats": {
      "ppg": "25.7",
      "rpg": "6.9"
  }
  "_version" : 3
}
```

このシナリオでは、サーバーにすでに存在するフィールド (interests、points、jersey) が欠落していることがわかります。また、マップ stats 内の ppg の値は編集中で、新しい値 rpg が追加され、apg は省略されています。Automerge は、省略されたフィールドを保持します (注意: フィールドを削除する予定の場合は、一致するバージョンでリクエストを再試行する必要があります)。したがって、それらは失われません。マップ内のフィールドにも同じルールが適用されるため、ppg への変更は拒否されますが、apg は保持され、新しいフィールド rpg が追加されます。その結果、サーバーに保存された項目は次のようになります。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
```

```
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a
 Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "stats": {
      "ppg": "35.4",
      "apg": "6.3",
      "rpg": "6.9"
  }
  "_version" : 9
}
```

lambda

いくつかの Lambda 解決戦略から選択できます。

- RESOLVE: 既存の項目を、レスポンスペイロードで提供された新しい項目に置き換えます。一度に 1 つの項目に対してのみ、同じオペレーションを再試行できます。現在、DynamoDB PutItem および UpdateItem でサポートされています。
- REJECT: ミューテーションを拒否し、GraphQL レスポンスで既存の項目のエラーを返します。現在、DynamoDB PutItem、UpdateItem、および DeleteItem でサポートされています。
- REMOVE: 既存の項目を削除します。現在、DynamoDB DeleteItem でサポートされています。

Lambda 呼び出しリクエスト

AWS AppSync DynamoDB リゾルバーは、 で指定された Lambda 関数を呼び出しますLambdaConflictHandlerArn。また、データソースに設定されたものと同じ service-role-arn を使用します。呼び出しのペイロードは以下の構造を持ちます。

```
{
    "newItem": { ... },
    "existingItem": {... },
    "arguments": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

各フィールドの定義は以下のようになります。

newItem

ミューテーションが成功した場合のプレビュー項目。

existingItem

現在、この項目は DynamoDB テーブルに存在しています。

arguments

GraphQL ミューテーションの引数です。

resolver

リ AWS AppSync ゾルバーに関する情報。

identity

呼び出し元に関する情報。API キーでアクセスする場合、このフィールドは null に設定されます。

ペイロードの例：

```
{
    "newItem": {
        "id": "1",
        "author": "Jeff",
        "title": "Foo Bar",
        "rating": 5,
        "comments": ["hello world"],
    },
    "existingItem": {
        "id": "1",
        "author": "Foo",
        "rating": 5,
        "comments": ["old comment"]
    },
    "arguments": {
        "id": "1",
        "author": "Jeff",
        "title": "Foo Bar",
        "comments": ["hello world"]
    },
    "resolver": {
        "tableName": "post-table",
        "awsRegion": "us-west-2",
        "parentType": "Mutation",
        "field": "updatePost"
    },
```

```
    "identity": {
         "accountId": "123456789012",
         "sourceIp": "x.x.x.x",
         "username": "AIDAAAAAAAAAAAAAAAAA",
         "userArn": "arn:aws:iam::123456789012:user/appsync"
    }
}
```

Lambda 呼び出しレスポンス

PutItem および UpdateItem の競合の解決

ミューテーション を RESOLVE する レスポンスは次の形式である必要があります。

```
{
    "action": "RESOLVE",
    "item": { ... }
}
```

item フィールドは、基になるデータソースの既存の項目を置き換えるために使用されるオブジェクトを表します。プライマリキーと同期メタデータは、item に含まれている場合は無視されます。

ミューテーション を REJECT する レスポンスは次の形式である必要があります。

```
{
    "action": "REJECT"
}
```

DeleteItem の競合の解決

項目を REMOVE します。レスポンスは次の形式である必要があります。

```
{
    "action": "REMOVE"
}
```

ミューテーション を REJECT する レスポンスは次の形式である必要があります。

```
{
    "action": "REJECT"
}
```

下記の Lambda 関数の例は、呼び出しを行うユーザーとリゾルバー名を確認します。jeffTheAdmin によって行われる場合、DeletePost リゾルバーのオブジェクトを削除 (REMOVE) するか、Update/Put リゾルバーの新しい項目との競合を解決 (RESOLVE) します。それ以外の場合、ミューテーションは REJECT です。

```
exports.handler = async (event, context, callback) => {
    console.log("Event: "+ JSON.stringify(event));

    // Business logic goes here.
    var response;
    if ( event.identity.user == "jeffTheAdmin" ) {
        let resolver = event.resolver.field;

        switch(resolver) {
            case "deletePost":
                response = {
                    "action" : "REMOVE"
                }
                break;

            case "updatePost":
            case "createPost":
                response = {
                    "action" : "RESOLVE",
                    "item": event.newItem
                }
                break;
            default:
                response = { "action" : "REJECT" };
        }
    } else {
        response = { "action" : "REJECT" };
    }

    console.log("Response: "+ JSON.stringify(response));
    return response;
}
```

エラー

以下は、競合解決プロセス中に発生する可能性のあるエラーのリストです。

ConflictUnhandled

競合の検出によりバージョンの不一致が検出され、競合ハンドラーによりミューテーションが拒否された。

例: オプティミスティック同時実行の競合ハンドラーを使用した競合の解決。または、Lambda 競合ハンドラーが REJECT として返される。

ConflictError

競合を解決しようとすると内部エラーが発生する。

例: Lambda 競合ハンドラーが誤った形式のレスポンスを返す。または、指定されたリソース LambdaConflictHandlerArn が見つからないため、Lambda 競合ハンドラーを呼び出せない。

MaxConflicts

競合解決のための最大再試行回数に達した。

例: 同じオブジェクトに対する同時リクエストが多すぎる。競合が解決される前に、オブジェクトは別のクライアントによって新しいバージョンに更新される。

BadRequest

クライアントがメタデータフィールド (_version、_ttl、_lastChangedAt、_deleted) を更新しようとした。

例: クライアントが、更新ミューテーションでオブジェクトの _version を更新しようとする。

DeltaSyncWriteError

差分同期レコードの書き込みに失敗した。

例: ミューテーションは成功したが、差分同期テーブルへの書き込み中に内部エラーが発生した。

InternalFailure

内部エラーが発生しました。

UnsupportedOperation

サポートされていないオペレーション '*X* '。データソースバージョニングは、次のオペレーション

(TransactGetItems 、PutItem 、BatchGetItem 、Scan、Query、GetItem 、DeleteItem 、UpdateItem 、S
のみをサポートします。

例: 競合の検出/解決を有効にした特定のトランザクションとバッチオペレーションの使用。以下のオペレーションは現在サポートされていません。

CloudWatch Logs

AWS AppSync API がログ記録設定をフィールドレベルログに設定enabledし、フィールドレベルログのログレベルを に設定して CloudWatch Logs を有効にしている場合ALL、 AWS AppSync は競合検出および解決情報をロググループに出力します。ログメッセージの形式については、競合の検出と同期のログ記録に関するドキュメントを参照してください。

のバージョニングされたデータソースでの DynamoDB 同期オペレーションの使用 AWS AppSync

バージョン管理されたデータソースは、DynamoDB テーブルからすべての結果を取得し、最後のクエリ (差分更新) 以降に変更されたデータのみを受け取る Sync オペレーションをサポートします。 AWS AppSync は、Syncオペレーションのリクエストを受信すると、リクエストで指定されたフィールドを使用して、ベーステーブルまたは差分テーブルにアクセスする必要があるかどうかを判断します。

- `lastSync` フィールドが指定されていない場合は、基本テーブルで `Scan` が実行されます。
- `lastSync` フィールドが指定され、値が`current moment - DeltaSyncTTL` の前にある場合は、基本テーブルで `Scan` が実行されます。
- `lastSync` フィールドが指定され、値が `current moment - DeltaSyncTTL` またはそれ以降にある場合は、差分テーブルで `Query` が実行されます。

AWS AppSync は、すべてのSyncオペレーションのレスポンスマッピングテンプレートに `startedAt`フィールドを返します。`startedAt` フィールドはエポックミリ秒単位の時刻ですが、Sync オペレーションが開始されれば、ローカルに保存して別のリクエストで使用することができます。ページ分割トークンがリクエストに含まれている場合、この値は、結果の最初のページのリクエストによって返されたものと同じになります。

Sync マッピングテンプレートの形式については、マッピングテンプレートのリファレンスを参照してください。

CloudWatch を使用した GraphQL API データのモニタリングとログ

CloudWatch メトリクスと CloudWatch ログを使用して、GraphQL API のログとデバッグを行うことができます。これらのツールにより、デベロッパーはパフォーマンスのモニタリング、問題のトラブルシューティング、GraphQL オペレーションの最適化を効果的に行うことができます。

CloudWatch メトリクスは、API のパフォーマンスと使用状況をモニタリングするための幅広いメトリクスを提供するツールです。これらのメトリクスは、2 つの主なカテゴリに分類されます。

1. 一般的な API メトリクス: これには、クライアントとサーバーのエラーを追跡するための `4XXError` および `5XXError`、レスポンスタイムを測定するための `Latency`、API コールの合計をモニタリングするための `Requests`、リソース使用状況を追跡するための `TokensConsumed` が含まれます。
2. リアルタイムサブスクリプションメトリクス: これらのメトリクスは、WebSocket 接続とサブスクリプションアクティビティに焦点を当てています。これには、接続リクエスト、成功した接続、サブスクリプション登録、メッセージ発行、アクティブな接続とサブスクリプションのメトリクスが含まれます。

このガイドでは、リゾルバーのパフォーマンス、データソースインタラクション、個々の GraphQL オペレーションに関するより詳細なデータを提供する拡張メトリクスも紹介しています。これらのメトリクスは、より深いインサイトを提供しますが、追加コストが発生します。

CloudWatch Logs は、GraphQL API のログ機能を有効にするツールです。ログは API の 2 つのレベルで設定できます。

1. リクエストレベルのログ: HTTP ヘッダー、GraphQL クエリ、オペレーションの概要、サブスクリプション登録など、全体的なリクエスト情報をキャプチャします。
2. フィールドレベルのログ: リクエストとレスポンスのマッピング、各フィールドのトレース情報など、個々のフィールド解決に関する詳細情報を提供します。

ログ記録の設定、ログエントリの解釈、トラブルシューティングと最適化にログデータを使用できます。 AWS AppSync には、クエリの実行、解析、検証、フィールド解決データを明らかにするさまざまなログタイプが用意されています。

セットアップと設定

GraphQL API で自動ログ記録を有効にするには、 AWS AppSync コンソールを使用します。

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
2. API] ページで、GraphQL API の名前を選択します。
3. API のホームページのナビゲーションペインで、[設定] を選択します。
4. [ログ記録] で以下を行います。
   a. [ログを有効にする] をオンにします。
   b. リクエストレベルの詳細なロギングを行うには、[詳細な内容を含める] のチェックボックスをオンにします。(オプション)
   c. フィールドリゾルバーのログレベルで、任意のフィールドレベルのログ記録レベル (なし、エラー、情報、デバッグ、またはすべて) を選択します (オプション) 。
   d. 既存のロールを作成または使用する で、新しいロールを選択して、 AWS AppSync が CloudWatch にログを書き込むことを許可する新しい AWS Identity and Access Management (IAM) を作成します。または、[既存のロール] を選択して、 AWS アカウント内の既存の IAM ロールの Amazon リソースネーム (ARN) を選択します。
5. [保存] を選択します。

手動での IAM ロールの設定

既存の IAM ロールを使用する場合、ロールは CloudWatch にログを書き込むために必要なアクセス許可を AWS AppSync に付与する必要があります。これを手動で設定するには、 AWS AppSync がログを書き込むときにロールを引き受けることができるように、サービスロール ARN を指定する必要があります。

IAM コンソールで、`AWSAppSyncPushToCloudWatchLogsPolicy` という名前の新しいポリシーを、以下の定義で作成します。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

次に、AWSAppSyncPushToCloudWatchLogsRole という名前で新しいロールを作成し、新しく作成したポリシーをこのロールにアタッチします。このロールの信頼関係を次のように編集します。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Effect": "Allow",
        "Principal": {
            "Service": "appsync.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
        }
    ]
}
```

ロール ARN をコピーし、 AWS AppSync GraphQL API のログ記録を設定するときに使用します。

CloudWatch メトリクス

CloudWatch メトリクスを使用して、HTTP ステータスコードやレイテンシーの原因となる特定のイベントを監視し、アラートを出すことができます。以下のメトリクスが出力されます。

メトリクスのリスト

4XXError

クライアントの設定が正しくないためにリクエストが無効になったために発生するエラー。通常、GraphQL を実行する外部の任意の場所で、これらのエラーが発生します。たとえば、このエラーは、リクエストに誤った JSON ペイロードまたは誤ったクエリが含まれている場合、サービスがスロットリングされている場合、または Auth 設定が誤って構成されている場合に発生する可能性があります。

単位: カウント これらのエラーの出現総数を取得するために、Sum 統計を使用します。

5XXError

GraphQL クエリの 実行中に発生したエラー。例えば、空のスキーマや不正確なスキーマに対してクエリを実行した場合に発生する可能性があります。また、Amazon Cognito ユーザープール ID または AWS リージョンが有効でない場合にも発生する可能性があります。また、リクエストの処理中に AWS AppSync で問題が発生した場合にもこれが発生することがあります。

単位: カウント これらのエラーの出現総数を取得するために、Sum 統計を使用します。

Latency

AWS AppSync がクライアントからリクエストを受信してからクライアントにレスポンスを返すまでの時間。エンドデバイスに到達するレスポンスに発生したネットワークレイテンシーは含まれません。

単位: ミリ秒 予測されるレイテンシーを評価するために Average 統計を使用します。

Requests

アカウント内のすべての API が処理したリクエスト (クエリ + ミューテーション) の数 (リージョン別)。

単位: カウント 特定のリージョンで処理されたすべてのリクエストの数。

TokensConsumed

トークンは、Request が使用するリソースの量 (処理時間と使用量) に基づいて Requests に割り当てられます。通常、それぞれの Request が 1 つのトークンを消費します。ただし、Request が大量のリソースを消費する場合には、必要に応じて追加のトークンが割り当てられます。

単位: カウント 特定のリージョンで処理されたリクエストに割り当てられるトークンの数。

`NetworkBandwidthOutAllowanceExceeded`

> **Note**
>
> AWS AppSync コンソールのキャッシュ設定ページで、キャッシュヘルスメトリクスオプションを使用すると、このキャッシュ関連のヘルスメトリクスを有効にできます。

スループットが集約された帯域幅制限を超えたためにドロップされたネットワークパケット。これは、キャッシュ設定のボトルネックを診断するのに役立ちます。データは、`appsyncCacheNetworkBandwidthOutAllowanceExceeded` メトリクスで `API_Id` を指定することで、特定の API に記録されます。

単位: カウント ID で指定された API の帯域幅制限を超えた後にドロップされたパケットの数。

`EngineCPUUtilization`

> **Note**
>
> AWS AppSync コンソールのキャッシュ設定ページで、キャッシュヘルスメトリクスオプションを使用すると、このキャッシュ関連のヘルスメトリクスを有効にできます。

Redis OSS プロセスに割り当てられた CPU 使用率 (パーセンテージ)。これは、キャッシュ設定のボトルネックを診断するのに役立ちます。データは、`appsyncCacheEngineCPUUtilization` メトリクスで `API_Id` を指定することで、特定の API に記録されます。

単位: パーセント。ID で指定された API の Redis OSS プロセスで現在使用されている CPU の割合。

リアルタイムサブスクリプション

すべてのメトリクスは、GraphQLAPIId という 1 つのディメンションで出力されます。これは、すべてのメトリクスが GraphQL API ID と結合されていることを意味します。次のメトリクスは、純粋な WebSocket を介した GrapQL サブスクリプションに関連しています。

メトリクスのリスト

`ConnectRequests`

成功した試行と失敗した試行の両方を含む AWS AppSync、 に対して行われた WebSocket 接続リクエストの数。

単位: カウント 接続リクエストの総数を取得するために Sum 統計を使用します。

`ConnectSuccess`

AWS AppSync への成功した WebSocket 接続の数。サブスクリプションなしで接続することは可能です。

単位: カウント 成功した接続の出現総数を取得するために、Sum 統計を使用します。

`ConnectClientError`

クライアント側のエラーが原因で AWS AppSync によって拒否された WebSocket 接続の数。これは、サービスがスロットリングされているか、承認設定が正しく構成されていないことを意味する可能性があります。

単位: カウント クライアント側の接続エラーの出現総数を取得するために、Sum 統計を使用します。

`ConnectServerError`

接続の処理中に AWS AppSync から発生したエラーの数。これは通常、予期しないサーバー側の問題が発生した場合に発生します。

単位: カウント サーバー側の接続エラーの出現総数を取得するために、Sum 統計を使用します。

`DisconnectSuccess`

AWS AppSync から正常に切断された WebSocket の数。

単位: カウント 成功した切断の出現総数を取得するために、Sum 統計を使用します。

`DisconnectClientError`

WebSocket 接続の切断中に AWS AppSync から発生したクライアントエラーの数。

単位: カウント 切断エラーの出現総数を取得するために、Sum 統計を使用します。

`DisconnectServerError`

WebSocket 接続の切断中に AWS AppSync から発生したサーバーエラーの数。

単位: カウント 切断エラーの出現総数を取得するために、Sum 統計を使用します。

`SubscribeSuccess`

WebSocket を介して AWS AppSync に正常に登録されたサブスクリプションの数。サブスクリプションがなくても接続することはできますが、接続せずにサブスクリプションを持つことはできません。

単位: カウント 成功したサブスクリプションの出現総数を取得するために、Sum 統計を使用します。

`SubscribeClientError`

クライアント側のエラーのために AWS AppSync によって拒否されたサブスクリプションの数。これは、JSON ペイロードが正しくない、サービスがスロットリングされている、または承認設定が正しく構成されていない場合に発生する可能性があります。

単位: カウント クライアント側のサブスクリプションエラーの出現総数を取得するために、Sum 統計を使用します。

`SubscribeServerError`

サブスクリプションの処理中に AWS AppSync から発生したエラーの数。これは通常、予期しないサーバー側の問題が発生した場合に発生します。

単位: カウント サーバー側のサブスクリプションエラーの出現総数を取得するために、Sum 統計を使用します。

`UnsubscribeSuccess`

正常に処理されたサブスクリプション解除リクエストの数。

単位: カウント 成功したサブスクリプション解除リクエストの出現総数を取得するために、Sum 統計を使用します。

`UnsubscribeClientError`

クライアント側のエラーが原因で AWS AppSync によって拒否されたサブスクリプション解除リクエストの数。

単位: カウント クライアント側のサブスクリプション解除リクエストのエラーの出現総数を取得するために、Sum 統計を使用します。

`UnsubscribeServerError`

サブスクリプション解除リクエストの処理中に AWS AppSync から発生したエラーの数。これは通常、予期しないサーバー側の問題が発生した場合に発生します。

単位: カウント サーバー側のサブスクリプション解除リクエストのエラーの出現総数を取得するために、Sum 統計を使用します。

`PublishDataMessageSuccess`

正常に発行されたサブスクリプションイベントメッセージの数。

単位: カウント 正常に発行されたサブスクリプションイベントメッセージの合計を取得するために、Sum 統計を使用します。

`PublishDataMessageClientError`

クライアント側のエラーのために発行に失敗したサブスクリプションイベントメッセージの数。

Unit: カウント クライアント側のサブスクリプションイベント発行エラーの出現総数を取得するために、Sum 統計を使用します。

`PublishDataMessageServerError`

サブスクリプションイベントメッセージの発行中に AWS AppSync から発生したエラーの数。これは通常、予期しないサーバー側の問題が発生した場合に発生します。

単位: カウント サーバー側のサブスクリプションイベント発行エラーの出現総数を取得するために、Sum 統計を使用します。

`PublishDataMessageSize`

発行されたサブスクリプションイベントメッセージのサイズ。

単位: バイト

`ActiveConnections`

サブスクリプションイベントメッセージの発行中に発生した AWS AppSync からのエラーの数。

単位: カウント 開かれている接続の合計数を取得するために、Sum 統計を使用します。

`ActiveSubscriptions`

クライアントからの同時サブスクリプション数 (1 分間)。

単位: カウント アクティブなサブスクリプションの合計数を取得するために、Sum 統計を使用します。

`ConnectionDuration`

接続が開いたままになる時間。

単位: ミリ秒 接続期間を評価するために Average 統計を使用します。

`OutboundMessages`

正常に公開された従量制メッセージの数。1 つの従量制メッセージは 5 kB の配信済みデータに相当します。

単位: カウント 正常に公開された従量制メッセージの総数を取得するために、Sum 統計を使用します。

`InboundMessageSuccess`

正常に処理されたインバウンドメッセージの数。ミューテーションによって呼び出されるサブスクリプションタイプごとに 1 つのインバウンドメッセージが生成されます。

単位: カウント 正常に処理されたインバウンドメッセージの総数を取得するために、Sum 統計を使用します。

`InboundMessageError`

サブスクリプションのペイロードサイズの制限である 240 kB を超えるなど、無効な API リクエストが原因で処理に失敗したインバウンドメッセージの数。

単位: カウント API 関連で処理に失敗したインバウンドメッセージの総数を取得するために、Sum 統計を使用します。

`InboundMessageFailure`

エラーが発生したために処理に失敗したインバウンドメッセージの数 AWS AppSync。

単位: カウント Sum 統計を使用して、 AWS AppSync関連する処理エラーが発生したインバウンドメッセージの総数を取得します。

`InboundMessageDelayed`

遅延インバウンドメッセージの数。インバウンドメッセージレートのクォータまたはアウトバウンドメッセージレートのクォータのいずれかが超過すると、インバウンドメッセージが遅延する可能性があります。

単位: カウント 合計の統計を使用して、遅延したインバウンドメッセージの総数を取得します。

`InboundMessageDropped`

ドロップされたインバウンドメッセージの数。インバウンドメッセージは、インバウンドメッセージレートのクォータまたはアウトバウンドメッセージレートのクォータのいずれかを超えた場合にドロップする可能性があります。

単位: カウント 合計の統計を使用して、ドロップされたインバウンドメッセージの総数を取得します。

`InvalidationSuccess`

`$extensions.invalidateSubscriptions()` とのミューテーションによって正常に無効 (購読解除) されたサブスクリプションの数。

単位: カウント サブスクライブが正常に解除されたサブスクリプションの総数を取得するには Sum 統計を使用します。

`InvalidationRequestSuccess`

正常に処理された無効化リクエストの数。

単位: カウント 正常に処理された無効化リクエストの総数を取得するために、Sum 統計を使用します。

`InvalidationRequestError`

無効な API リクエストにより処理に失敗した無効化リクエストの数。

単位: カウント API 関連で処理に失敗した無効化リクエストの総数を取得するために、Sum 統計を使用します。

`InvalidationRequestFailure`

エラーにより処理に失敗した無効化リクエストの数 AWS AppSync。

単位: カウント Sum 統計を使用して、 AWS AppSync関連する処理に失敗した無効化リクエストの総数を取得します。

`InvalidationRequestDropped`

無効化リクエストのクォータを超えたときにドロップされた無効化リクエストの数。

単位: カウント ドロップされた無効化リクエストの総数を取得するために、Sum 統計を使用します。

インバウンドメッセージとアウトバウンドメッセージの比較

ミューテーションを実行すると、そのミューテーションの @aws_subscribe ディレクティブを含むサブスクリプションフィールドが呼び出されます。サブスクリプションを呼び出すたびに、1 つのインバウンドメッセージが生成されます。例えば、2 つのサブスクリプションフィールドが @aws_subscribe で同じミューテーションを指定している場合、そのミューテーションが呼び出されると 2 つのインバウンドメッセージが生成されます。

1 つのアウトバウンドメッセージは、WebSocket クライアントに配信される 5 kB のデータに相当します。例えば、15 kB のデータを 10 個のクライアントに送信すると、アウトバウンドメッセージは 30 件になります (15 kB × 10 クライアント ÷ 1 メッセージあたり 5 kB = 30 メッセージ)。

インバウンドメッセージまたはアウトバウンドメッセージのクォータ引き上げをリクエストできます。詳細については、「AWS 参考文献」の「AWS AppSync エンドポイントとクォータ」、「Service Quotas クォータユーザーガイド」の「クォータ増加のリクエスト」の手順を参照してください。

拡張メトリクス

拡張メトリクスは、 AWS AppSync リクエスト数やエラー数、レイテンシー、キャッシュヒット/ミスなど、API の使用状況とパフォーマンスに関する詳細なデータを生成します。拡張メトリクスデータはすべて CloudWatch アカウントに送信され、送信されるデータのタイプを設定できます。

> **Note**
>
> 拡張メトリクスを使用する場合、追加料金が適用されます。詳細については、「Amazon CloudWatch の料金」の「詳細モニタリング」の料金レベルを参照してください。

これらのメトリクスは、 AWS AppSync コンソールのさまざまな設定ページで確認できます。API 設定ページで、拡張メトリクスセクションを使用すると、次の項目を有効または無効にできます。

リゾルバーメトリクスの動作: これらのオプションは、リゾルバーの追加メトリクスの収集方法を制御します。フルリクエストリゾルバーメトリクス (リクエスト内のすべてのリゾルバーに対して有効になっているメトリクス) またはリゾルバーごとのメトリクス (設定が有効になっているリゾルバーに対してのみ有効になっているメトリクス) を有効にすることができます。以下のオプションが利用できます。

リゾルバーメトリクスの動作リスト

`GraphQL errors per resolver (GraphQLError)`

リゾルバーごとに発生した GraphQL エラーの数。

メトリクスディメンション: API_Id、Resolver

単位: カウント

`Requests per resolver (Request)`

リクエスト中に発生した呼び出しの数。これはリゾルバーごとに記録されます。

メトリクスディメンション: API_Id、Resolver

単位: カウント

`Latency per resolver (Latency)`

リゾルバーの呼び出しを完了する時間。レイテンシーはミリ秒単位で測定され、リゾルバーごとに記録されます。

メトリクスディメンション: API_Id、Resolver

単位: ミリ秒

`Cache hits per resolver (CacheHit)`

リクエスト中のキャッシュヒットの数。これは、キャッシュが使用されている場合にのみ出力されます。キャッシュヒットはリゾルバーごとに記録されます。

メトリクスディメンション: API_Id、Resolver

単位: カウント

`Cache misses per resolver (CacheMiss)`

リクエスト中のキャッシュミスの数。これは、キャッシュが使用されている場合にのみ出力されます。キャッシュミスはリゾルバーごとに記録されます。

メトリクスディメンション: API_Id、Resolver

単位: カウント

データソースメトリクスの動作: これらのオプションは、データソースの追加メトリクスの収集方法を制御します。フルリクエストデータソースメトリクス (リクエスト内のすべてのデータソースで有効になっているメトリクス) またはデータソースごとのメトリクス (設定が有効になっているデータソースでのみ有効になっているメトリクス) を有効にすることができます。以下のオプションが利用できます。

データソースメトリクスの動作リスト

Requests per data source (Request)

リクエスト中に発生した呼び出しの数。リクエストはデータソースごとに記録されます。完全なリクエストが有効になっている場合、各データソースは CloudWatch に独自のエントリを持ちます。

メトリクスディメンション: API_Id、Datasource

単位: カウント

Latency per data source (Latency)

データソースの呼び出しを完了する時間。レイテンシーはデータソースごとに記録されます。

メトリクスディメンション: API_Id、Datasource

単位: ミリ秒

Errors per data source (GraphQLError)

データソースの呼び出し中に発生したエラーの数。

メトリクスディメンション: API_Id、Datasource

単位: カウント

オペレーションメトリクス: GraphQL オペレーションレベルのメトリクスを有効にします。

オペレーションメトリクスの動作リスト

`Requests per operation (Request)`

指定された GraphQL オペレーションが呼び出された回数。

メトリクスディメンション: `API_Id`、`Operation`

単位: カウント

`GraphQL errors per operation (GraphQLError)`

指定された GraphQL オペレーション中に発生した GraphQL エラーの数。

メトリクスディメンション: `API_Id`、`Operation`

単位: カウント

CloudWatch ログ

任意の新規または既存の GraphQL API でリクエストレベルおよびフィールドレベルの 2 つのタイプでログ記録するように設定できます。

リクエストレベルログ

リクエストレベルのロギング (詳細な内容を含む) を設定すると、次の情報がログに記録されます。

- 消費されたトークンの数。
- リクエストとレスポンスの HTTP ヘッダー
- リクエストで実行中の GraphQL クエリ
- オペレーション全体の要約
- 登録された、新規および既存の GraphQL サブスクリプション

フィールドレベルログ

フィールドレベルのロギングを設定すると、以下の情報がログに記録されます。

- 各フィールドに対してソースと引数で生成されたリクエストマッピング

- 各フィールドの、変換されたレスポンスマッピングで、対象フィールドを解決した結果のデータが含まれます。
- 各フィールドのトレース情報

ログ記録を有効にすると、 AWS AppSync は CloudWatch Logs を管理します。このプロセスには、ロググループとログストリームの作成、およびログとログストリームへのレポートが含まれます。

GraphQL API でログ記録を有効にしてリクエストを行うと、 AWS AppSync はロググループの下にロググループとログストリームを作成します。ロググループの名前は `/aws/appsync/apis/{graphql_api_id}` 形式に従います。各ロググループ内で、ログはログストリームにさらに分割されます。これらは、ログデータがレポートされるときに Last Event Time によって順序付けされます。

すべてのログイベントは、対象リクエストに x-amzn-RequestId でタグ付けされています。これにより、リクエストに関してログに記録されたすべての情報を取得するために CloudWatch のログイベントをフィルター処理できます。すべての GraphQL AWS AppSync リクエストの応答ヘッダから RequestId を取得することができます。

フィールドレベルのログ記録は次のログレベルで設定されます。

- なし - フィールドレベルのログはキャプチャされません。
- Error - エラーカテゴリにあるフィールドについてのみ、次の情報をログに記録します。
    - サーバーレスポンスのエラーセクション
    - フィールドレベルエラー
    - エラーフィールドに対して解決された、生成リクエスト / レスポンス関数
- 情報 - 情報カテゴリとエラーカテゴリにあるフィールドについてのみ、次の情報をログに記録します。
    - 情報レベルのメッセージ
    - `$util.log.info` および を介して送信されるユーザーメッセージ `console.log`
    - フィールドレベルのトレースログとマッピングログは表示されません。
    - フィールドレベルのログ記録が詳細コンテンツを含めて `INFO`以上に設定されている場合、は変換されたマッピングテンプレートのログ記録メッセージ AWS AppSync を追加します。これには、変換されたマッピングテンプレートに追加された情報、またはリゾルバーまたは関数が実行した JavaScript コードの出力が含まれます。パスワードや認可ヘッダーなどの機密情報をダウンストリームデータソースに送信し、その情報をログに含めたくない場合は、使用しないでください。

- デバッグ - デバッグ、情報、エラーカテゴリにあるフィールドについてのみ、次の情報を口グに記録します。
    - デバッグレベルのメッセージ
    - `$util.log.info`、、`$util.log.debug`、`console.log`および を介して送信されるユーザーメッセージ `console.debug`
    - フィールドレベルのトレースログとマッピングログは表示されません。
- ALL - 次の情報が、クエリのすべてのフィールドに記録されます。
    - フィールドレベルのトレース情報
    - 各フィールドで解決された生成されたリクエスト/レスポンス関数

モニタリングのメリット

ログおよびメトリクスを使用して、GraphQL クエリを特定、トラブルシューティング、最適化できます。たとえば、クエリの各フィールドに記録されたトレース情報を使用して、レイテンシーの問題をデバッグできます。これを示すため、GraphQL クエリで 1 つまたは複数のリゾルバーをネストして使用しているとします。CloudWatch Logs のフィールドオペレーションのサンプルは次のようになります。

```
{
    "path": [
        "singlePost",
        "authors",
        0,
        "name"
    ],
    "parentType": "Post",
    "returnType": "String!",
    "fieldName": "name",
    "startOffset": 416563350,
    "duration": 11247
}
```

これは GraphQL スキーマに対応します。次のようになります。

```
type Post {
  id: ID!
  name: String!
  authors: [Author]
```

```
}

type Author {
  id: ID!
  name: String!
}

type Query {
  singlePost(id:ID!): Post
}
```

上記のログの結果で、path は、`singlePost()` という名前のクエリを実行して返されたデータの単一項目を示します。この例では、最初のインデックス (0) の name フィールドを表します。startOffset は、GraphQL クエリオペレーションの開始からのオフセットです。duration は、フィールドを解決するのにかかる合計時間です。これらの値は、特定のデータソースからデータの実行が予測より遅い理由、または特定のフィールドがクエリ全体の処理速度を低下することのトラブルシューティングに役立つ可能性があります。例えば、Amazon DynamoDB テーブルのプロビジョニングスループットを向上させたり、オペレーション全体のパフォーマンスが低下する原因となっている特定のフィールドをクエリから削除したりすることができます。

2019 年 5 月 8 日現在、 AWS AppSync は完全に構造化された JSON としてログイベントを生成します。これにより、CloudWatch Logs Insights や Amazon OpenSearch Service などのログ分析サービスを使用して、GraphQL リクエストのパフォーマンスとスキーマフィールドの使用傾向を把握できるようになります。たとえば、パフォーマンスの問題の根本原因となっている可能性のある高いレイテンシーが発生しているリゾルバーの特定を簡単に行えます。また、スキーマ内で最も使用頻度の低いフィールドを特定し、GraphQL フィールドを廃止する場合の影響を評価することもできます。

競合の検出と同期ロギング

AWS AppSync API に、フィールドリゾルバーのログレベルが All に設定されている CloudWatch Logs へのログ記録がある場合、 AWS AppSync は競合検出と解決情報をロググループに出力します。これにより、 AWS AppSync API が競合にどのように応答したかを詳細に把握できます。応答を解釈しやすくするため、ログには以下の情報が記録されます。

メトリクスのリスト

`conflictType`

バージョンの不一致またはお客様が指定した条件が原因で競合が発生したかどうかを詳細に示します。

`conflictHandlerConfigured`

リクエスト時にリゾルバーで設定された競合ハンドラーを示します。

`message`

競合をどのように検出し、解決したかに関する情報を提供します。

`syncAttempt`

リクエストを最終的に拒否する前に、サーバーがデータを同期するために試行した回数。

`data`

競合ハンドラーが `Automerge` に設定されていた場合、このフィールドは `Automergee` が各フィールドに対してどのような決定を行ったかを示すために入力されます。提供されるアクションは、次のとおりです。

- REJECTED - `Automerge` がサーバーの値を優先して受信フィールド値を拒否した場合。
- ADDED - サーバーに既存の値がないために、`Automerge` が着信フィールドに追加した場合。
- APPENDED - `Automerge` がサーバーに存在するリストの値に着信値を追加した場合。
- MERGED - `Automerge` が受信した値をサーバーに存在するセットの値にマージした場合。

トークン数を使用してリクエストを最適化する

1,500 KB 秒以下のメモリと vCPU 時間を消費するリクエストには、1 つのトークンが割り当てられます。リソース消費量が 1,500 KB 秒を超えるリクエストには、追加のトークンが渡されます。たとえば、リクエストが 3,350 KB 秒を消費する場合、 AWS AppSync はリクエストに 3 つのトークン (次の整数値に切り上げ) を割り当てます。デフォルトでは、 AWS AppSync はデプロイ先の AWS リージョンに応じて、1 秒あたり最大 5,000 または 10,000 個のリクエストトークンをアカウントの APIs に割り当てます。各 API が 1 秒あたり平均 2 つのトークンを使用する場合、1 秒あたりのリクエスト数は 2,500 または 5,000 に制限されます。1 秒あたりに割り当てられた量よりも多くのトークンが必要な場合は、リクエストを送信してリクエストトークンのレートのデフォルトクォータを増やすことができます。詳細については、「AWS 全般のリファレンス ガイド」の「AWS AppSync エンドポイントとクォータ」、「Service Quotas クォータユーザーガイド」の「クォータ増加のリクエスト」を参照してください。

リクエストあたりのトークン数が多いということは、リクエストを最適化して API のパフォーマンスを向上させるチャンスがあることを示している可能性があります。リクエスト 1 回あたりのトークン数を増やす要因には次のようなものがあります。

- GraphQL スキーマのサイズと複雑さ。
- リクエストとレスポンスのマッピングテンプレートの複雑さ。
- 1 回のリゾルバー呼び出し回数。
- リゾルバーから返されたデータ転送量。
- ダウンストリームデータソースのレイテンシー。
- (parallel 呼び出しやバッチ呼び出しとは対照的に) 連続したデータソース呼び出しを必要とするスキーマとクエリの設計。
- ロギング設定、特にフィールドレベルおよび詳細なログコンテンツ。

> **Note**
>
> AWS AppSync メトリクスとログに加えて、クライアントはレスポンスヘッダー を介してリクエストで消費されたトークンの数にアクセスできますx-amzn-appsync-TokensConsumed。

ログのサイズ制限

デフォルトでは、ログ記録が有効になっている場合、 AWS AppSync はリクエストごとに最大 1 MB のログを送信します。このサイズを超えるログは切り捨てられます。ログサイズを減らすには、フィールドレベルのログの ERROR ログレベルを選択し、VERBOSE ログを無効にするか、不要な場合はフィールドレベルのログを完全に無効にします。ALL ログレベルの代替として、拡張メトリクスを使用して特定のリゾルバー、データソース、または GraphQL オペレーションに関するメトリクスを取得したり、AppSync が提供するログユーティリティを使用して必要な情報のみをログに記録したりできます。

ログタイプリファレンス

RequestSummary

- requestId: リクエストの一意の識別子。
- graphQLAPIId: リクエスト元の GraphQL API の ID。
- statusCode: HTTP ステータスコードのレスポンス。
- latency: リクエストのエンドツーエンドのレイテンシー (ナノ秒単位、整数)。

```
{
     "logType": "RequestSummary",
     "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
     "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4",
     "statusCode": 200,
     "latency": 242000000
}
```

ExecutionSummary

- requestId: リクエストの一意の識別子。
- graphQLAPIId: リクエスト元の GraphQL API の ID。
- startTime: リクエストの GraphQL 処理の開始タイムスタンプ (RFC 3339 形式)。
- endTime: リクエストの GraphQL 処理の終了タイムスタンプ (RFC 3339 形式)。
- duration: GraphQL 処理の合計経過時間 (ナノ秒単位、整数)。
- version: ExecutionSummary のスキーマバージョン。
- parsing:
    - startOffset: 解析の開始オフセット (呼び出しを基準としたナノ秒単位、整数)。
    - duration: 解析にかかった時間 (ナノ秒単位、整数)。
- validation:
    - startOffset: 検証の開始オフセット (呼び出しを基準としたナノ秒単位、整数)。
    - duration: 検証の実行にかかった時間 (ナノ秒単位、整数)。

```
{
    "duration": 217406145,
    "logType": "ExecutionSummary",
    "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
    "startTime": "2019-01-01T06:06:18.956Z",
    "endTime": "2019-01-01T06:06:19.174Z",
    "parsing": {
        "startOffset": 49033,
        "duration": 34784
    },
    "version": 1,
    "validation": {
        "startOffset": 129048,
        "duration": 69126
```

```
    },
    "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4"
}
```

トレース

- requestId: リクエストの一意の識別子。
- graphQLAPIId: リクエスト元の GraphQL API の ID。
- startOffset: フィールドの解決の開始オフセット (呼び出しを基準としたナノ秒単位、整数)。
- duration: フィールドの解決にかかった時間 (ナノ秒単位、整数)。
- fieldName: 解決されるフィールドの名前。
- parentType: 解決されるフィールドの親タイプ。
- returnType: 解決されるフィールドの戻り値の型。
- path: レスポンスのルートから解決されるフィールドまでのパスセグメントのリスト。
- resolverArn: フィールドの解決に使用されるリゾルバーの ARN。ネストされたフィールドには存在しない場合があります。

```
{
    "duration": 216820346,
    "logType": "Tracing",
    "path": [
        "putItem"
    ],
    "fieldName": "putItem",
    "startOffset": 178156,
    "resolverArn": "arn:aws:appsync:us-east-1:111111111111:apis/
pmo28inf75eepg63qxq4ekoeg4/types/Mutation/fields/putItem",
    "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
    "parentType": "Mutation",
    "returnType": "Item",
    "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4"
}
```

CloudWatch Logs Insights を使用したログの分析

以下は、GraphQL オペレーションのパフォーマンスとヘルスに関する実用的な洞察を得るために実行できるクエリの例です。これらの例は、CloudWatch Logs Insights コンソールでサンプルクエリ

として利用できます。CloudWatch コンソールで、Logs Insights を選択し、GraphQL API の AWS AppSync ロググループを選択し、サンプルAWS AppSync クエリでクエリを選択します。

以下のクエリは、レイテンシーが最大の上位 10 個の GraphQL リクエストを返します。

```
filter @message like "Tokens Consumed"
| parse @message "* Tokens Consumed: *" as requestId, tokens
| sort tokens desc
| display requestId, tokens
| limit 10
```

以下のクエリは、レイテンシーが最大の上位 10 個のリゾルバーを返します。

```
  fields resolverArn, duration
| filter logType = "Tracing"
| limit 10
| sort duration desc
```

以下のクエリは、呼び出し頻度が最も高いリゾルバーを返します。

```
  fields ispresent(resolverArn) as isRes
| stats count() as invocationCount by resolverArn
| filter isRes and logType = "Tracing"
| limit 10
| sort invocationCount desc
```

以下のクエリは、マッピングテンプレートのエラーが最も多いリゾルバーを返します。

```
  fields ispresent(resolverArn) as isRes
| stats count() as errorCount by resolverArn, logType
| filter isRes and (logType = "RequestMapping" or logType = "ResponseMapping") and
 fieldInError
| limit 10
| sort errorCount desc
```

以下のクエリは、リゾルバーのレイテンシー統計を返します。

```
  fields ispresent(resolverArn) as isRes
| stats min(duration), max(duration), avg(duration) as avg_dur by resolverArn
| filter isRes and logType = "Tracing"
```

```
| limit 10
| sort avg_dur desc
```

以下のクエリは、フィールドのレイテンシー統計を返します。

```
  stats min(duration), max(duration), avg(duration) as avg_dur
  by concat(parentType, '/', fieldName) as fieldKey
| filter logType = "Tracing"
| limit 10
| sort avg_dur desc
```

CloudWatch Logs Insights クエリの結果は CloudWatch ダッシュボードにエクスポートできます。

OpenSearch Service を使用してログを分析する

Amazon OpenSearch Service を使用して AWS AppSync ログを検索、分析、視覚化し、パフォーマンスのボトルネックと運用上の問題の根本原因を特定できます。レイテンシーが最大でエラーのあるリゾルバーを特定できます。さらに、OpenSearch を使用して、可視化に優れたダッシュボードを作成できます。OpenSearch Dashboards は、OpenSearch Service で利用可能なオープンソースのデータ可視化および探索ツールです。OpenSearch ダッシュボードを使用すると、GraphQL オペレーションのパフォーマンスとヘルスを継続的にモニタリングできます。例えば、GraphQL リクエストの P90 レイテンシーを可視化し、各リゾルバーの P90 レイテンシーにドリルダウンできるダッシュボードを作成できます。

OpenSearch Service を使用するときは、OpenSearch インデックスを検索するためのフィルターパターンとして「「cwl*」を使用してください。OpenSearch Service は、CloudWatch Logs からストリーミングされるログに「cwl-」というプレフィックスを付けてインデックスを作成します。 AWS AppSync API ログを OpenSearch Service に送信された他の CloudWatch ログと区別するには、検索`graphQLAPIID.keyword=`*`YourGraphQLAPIID`*に のフィルター式を追加することをお勧めします。

ログ形式の移行

2019 年 5 月 8 日以降に AWS AppSync が生成するログイベントは、完全に構造化された JSON としてフォーマットされます。2019 年 5 月 8 日より前の GraphQL リクエストを分析する場合は、GitHub Sample で入手可能なスクリプトを使用して、古いログを完全に構造化された JSON に移行できます。2019 年 5 月 8 日より前のログ形式を使用する必要がある場合は、[Type (タイプ)] を [Account Management (アカウント管理)] に、[Category (カテゴリ)] を [General Account Question (一般的なアカウント質問)] に設定して、サポートチケットを作成します。

次にこれらのメトリクスフィルターを使用して、ログデータをグラフの作成やアラームの設定に利用できる数値の CloudWatch メトリクスに変換します。

AWS X-Ray を使用して AWS AppSync でリクエストをトレースする

を使用してAWS X-Ray、 AWS AppSync で実行されるリクエストをトレースできます。X-Ray が利用可能なすべての AWS リージョンで AWS AppSync で X-Ray を使用できます。X-Ray は、GraphQL リクエスト全体の詳細な概要を提供します。これにより、API とその基盤となるリゾルバーとデータソース内のレイテンシーを分析できます。X-Ray サービスマップを使用して、X-Ray と統合された AWS サービスを含むリクエストのレイテンシーを表示できます。また、サンプリングルールを設定することで、X-Ray で記録するリクエストとサンプリングレートを基準に応じて指定できます。

X-Ray でのサンプリングの詳細については、「AWS X-Ray コンソールでのサンプリングルールの設定」を参照してください。

セットアップと設定

GraphQL API の X-Ray トレースは、 AWS AppSync コンソールから有効にできます。

1. AWS AppSync コンソールにサインインします。
2. ナビゲーションパネルから [設定] を選択します。
3. X-Ray で、[X-Ray を有効にする] をオンにします。
4. [Save] を選択します。X-Ray が API に対して有効になりました。

AWS CLI または を使用している場合は AWS CloudFormation、新しい AWS AppSync API を作成するときに X-Ray トレースを有効にするか、 `xrayEnabled`プロパティを に設定して既存の AWS AppSync API を更新することもできます`true`。

AWS AppSync API で X-Ray トレースを有効にすると、適切なアクセス許可を持つ AWS Identity and Access Management サービスにリンクされたロールがアカウントに自動的に作成されます。これにより、 AWS AppSync は安全な方法で X-Ray にトレースを送信できます。

X-Ray で API をトレースする

サンプリング

サンプリングルールを使用することで、コードを変更または再デプロイすることなく、その場で、AWS AppSync で記録するレコードの量を制御したり、サンプリング動作を変更したりできます。たとえば、このルールは、API ID 3n572shhcpfokwhdnq1ogu59v6 を持つ GraphQL API へのリクエストをサンプリングします。

- ルール名 — `test-sample`
- 優先度 — `10`
- リザーバのサイズ — `10`
- 固定レート — `10`
- サービス名 — *
- サービスタイプ — `AWS::AppSync::GraphQLAPI`
- HTTP メソッド — *
- リソース ARN — `arn:aws:appsync:us-west-2:123456789012:apis/3n572shhcpfokwhdnq1ogu59v6`
- ホスト — *

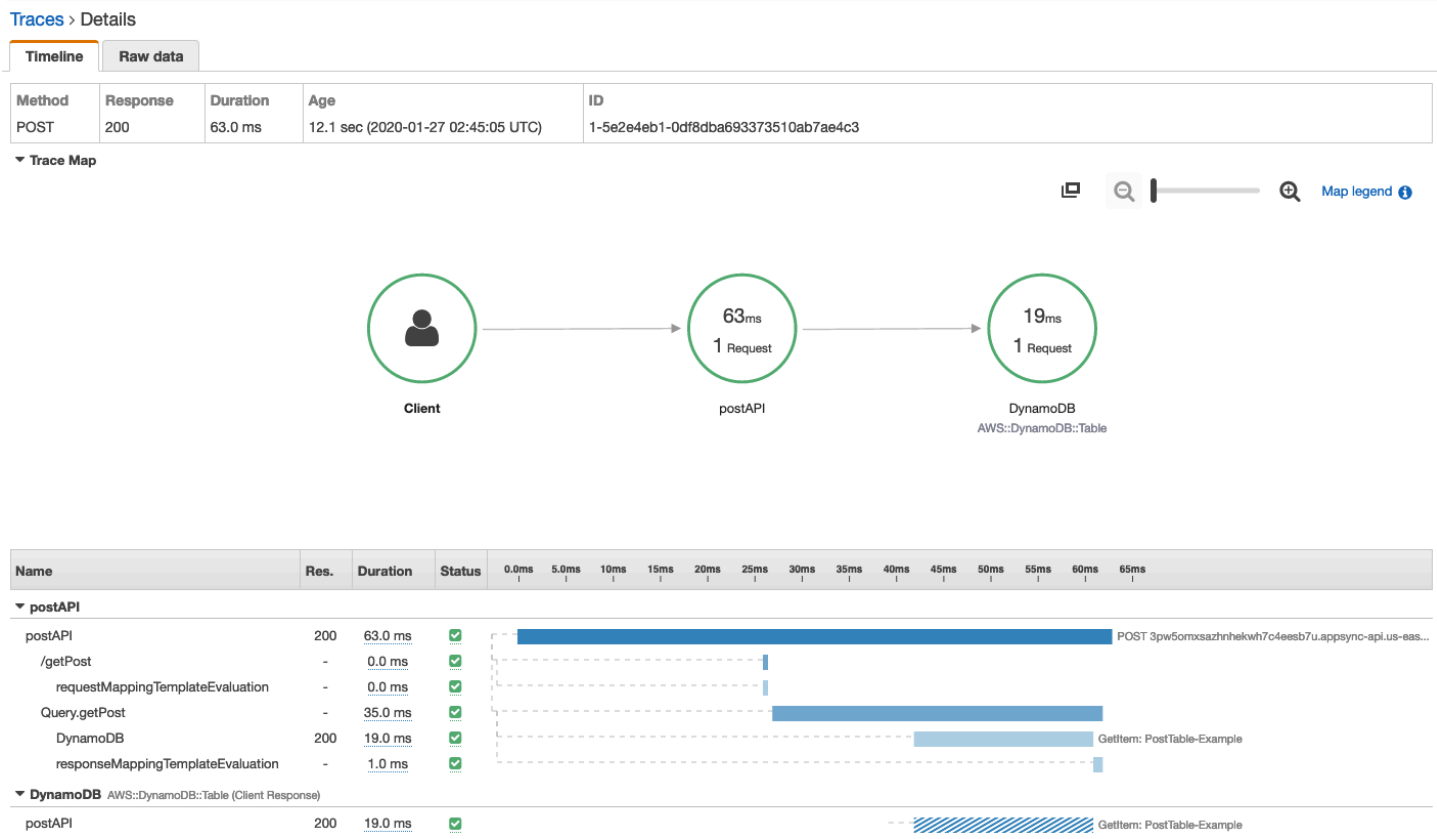
トレースについて

GraphQL API の X-Ray トレースを有効にすると、X-Ray トレース詳細ページを使用して、API に対するリクエストの詳細なレイテンシー情報を調べることができます。次に、この特定のリクエストのサービスマップとともにトレースビューを表示する例を示します。リクエストは Post タイプを使用して `postAPI` と呼ばれる API に対して行われました。そのデータは `PostTable-Example` という名前の Amazon DynamoDB テーブルに含まれています。

次のトレースイメージは、次の GraphQL クエリに対応しています。

```
query getPost {
    getPost(id: "1") {
      id
      title
    }
}
```

getPost クエリのリゾルバーは、基盤になる DynamoDB データソースを使用します。次のトレースビューには、 DynamoDB への呼び出しと、クエリの実行のさまざまな部分のレイテンシーが表示されます。



- 上の図では、/getPost は、解決される要素への完全なパスを表します。この場合、getPost はルート Query タイプのフィールドであるため、パスのルートの直後に表示されます。
- requestMappingTemplateEvaluation は、 AWS AppSync によってクエリ内のこの要素のリクエストマッピングテンプレートを評価するのに費やされた時間を表します。
- Query.getPost は、型とフィールド (Type.field 形式) を表します。API の構造とトレースされるリクエストに応じて、複数のサブセグメントを含めることができます。
  - DynamoDB は、このリゾルバーにアタッチされているデータソースを表します。これには、DynamoDB へのネットワーク呼び出しがフィールドを解決するためのレイテンシーが含まれます。
  - responseMappingTemplateEvaluation は、 AWS AppSync によってクエリ内のこの要素のレスポンスマッピングテンプレートを評価するのに費やされた時間を表します。

X-Ray でトレースを表示する場合、サブセグメントを選択し、詳細ビューを調べることで、 AWS AppSync セグメントのサブセグメントに関する追加のコンテキスト情報とメタデータ情報を取得できます。

深くネストされたクエリや複雑なクエリについては、セグメントAWS X-Ray ドキュメントで定義されているように、 AWS AppSync によって X-Ray に配信されるセグメントは、セグメントドキュメントで許可される最大サイズよりも大きくなる可能性があることに注意してください。X-Ray では、制限を超えるセグメントは表示されません。

を使用した AWS AppSync API コールのログ記録 AWS CloudTrail

AWS AppSync は、 のユーザー AWS CloudTrail、ロール、またはサービスによって実行されたアクションを記録する AWS サービスである と統合されています AWS AppSync。CloudTrail は、 のすべての API コールをイベント AWS AppSync 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 AppSync コンソールからの呼び出しと、 AWS AppSync APIs へのコード呼び出しが含まれます。CloudTrail によって収集された情報を使用して、リクエストの実行先 AWS AppSync、リクエスタの IP アドレス、リクエストの実行者、リクエストの実行日時などの詳細を確認できます。

証跡を作成して、イベントを含む Amazon Simple Storage Service (Amazon S3) バケットへの CloudTrail イベントの継続的な配信を有効にすることができます AWS AppSync。証跡を設定しない場合でも、CloudTrail コンソールで最新のイベントを表示できます。

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。

AWS AppSync CloudTrail の情報

CloudTrail は、 AWS アカウントの作成時にアカウントで有効になります。イベント履歴の CloudTrail コンソールで、 AWS アカウントnの最近のイベントを、表示、検索、ダウンロードできます。詳細については、「AWS CloudTrail ユーザーガイド」の「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。

のイベントなど、 AWS アカウント内のイベントの継続的な記録については AWS AppSync、証跡を作成します。デフォルトでは、コンソールで証跡を作成すると、すべての AWS リージョンに証跡が適用されます。証跡は、 AWS パーティション内のすべてのリージョンからのイベントをログに記録し、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、AWS CloudTrail ユーザーガイドで次を参照してください。

- AWS アカウントの証跡の作成

- AWS CloudTrail ログとのサービス統合
- CloudTrail の Amazon SNS 通知の設定
- CloudTrail ログファイルの複数のリージョンからの受け取り
- 複数のアカウントから CloudTrailログファイルを受け取る

CloudTrail はすべての AWS AppSync API オペレーションをログに記録します。たとえば、`CreateGraphqlApi`、`CreateDataSource`、および `ListResolvers` の各 API を呼び出すと、CloudTrail ログファイルにエントリが生成されます。これらのオペレーションおよびその他のオペレーションは、に記載されています。「AWS AppSync API リファレンス」に記載されています。

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。この ID 情報は以下のことを確認するのに役立ちます。

- リクエストが root または AWS Identity and Access Management (IAM) ユーザー認証情報を使用して行われたかどうか。
- リクエストがロールまたはフェデレーションユーザーのテンポラリなセキュリティ認証情報を使用して行われたかどうか。
- リクエストが別の AWS サービスによって行われたかどうか。

詳細については、「AWS CloudTrail ユーザーガイド」の「CloudTrail userIdentity Element」を参照してください。

AWS AppSync CloudTrail のデータイベント

データイベントでは、リソース上またはリソース内で実行されるリソースオペレーション (Amazon S3 オブジェクトの読み取りまたは書き込みなど) についての情報が得られます。これらのイベントは、データプレーンオペレーションとも呼ばれます。データイベントは、多くの場合、高ボリュームのアクティビティです。デフォルトでは、CloudTrail はデータイベントをログ記録しません。CloudTrail [イベント履歴] にはデータイベントは記録されません。

追加の変更がイベントデータに適用されます。CloudTrail の料金の詳細については、「AWS CloudTrail の料金」を参照してください。

CloudTrail コンソール AWS CLI、または CloudTrail API オペレーション (クエリ、ミューテーション、サブスクリプションオペレーションなど、リアルタイム WebSocket エンドポイントにオペレーションを接続しますが、リアルタイム WebSocket エンドポイント経由で送信されるメッセージは送信しません) を使用して、`AWS::AppSync::GraphQLApi`リソースタイプのデータイベントをログ

に記録できます。データイベントをログに記録する方法の詳細については、「AWS CloudTrail ユーザーガイド」の「AWS Management Consoleを使用したデータイベントのログ記録」および「AWS Command Line Interfaceを使用したデータイベントのログ記録」を参照してください。

次の表に、データイベントを口グ記録できる AWS AppSync リソースタイプを示します。データイベントタイプ (コンソール) 列には、CloudTrail コンソールのデータイベントタイプリストから選択する値が表示されます。resources.type 値列には、 AWS CLI または CloudTrail APIs を使用して高度なイベントセレクタを設定するときに指定する `resources.type`値が表示されます。CloudTrail に記録されたデータ API 列には、リソース タイプの CloudTrail にログ記録された API コールが表示されます。

データイベントタイプ (コンソール)	resources.type 値	CloudTrail にログ記録されたデータ API
AppSync GraphQL	`AWS::AppSync::GraphQLApi`	GraphQL

`eventName`、`readOnly`、および `resources.ARN` フィールドでフィルタリングして、自分にとって重要なイベントのみをログに記録するように高度なイベントセレクタを設定できます。オブジェクトの詳細については、「AWS CloudTrail API リファレンス」の「AdvancedFieldSelector」を参照してください。

```
[
  {
    "name": "Only 1 AppSync API",
    "fieldSelectors": [
      {
        "field": "eventCategory",
        "equals": [
          "Data"
        ]
      },
      {
        "field": "resources.type",
        "equals": [
          "AWS::AppSync::GraphQLApi"
        ]
      },
      {
```

```
        "field": "resources.ARN",
        "equals": [
          "arn:aws:appsync:us-east-1:111122223333:apis/YourGraphQLApiId"
        ]
      }
    ]
  }
]
```

AWS AppSync ログファイルエントリについて

CloudTrail は、1 つ以上のログエントリがあるログファイルとしてイベントを送信します。各イベントは任意の送信元からの単一のリクエストを表し、リクエストされたオペレーション、オペレーションの日時、リクエストパラメータなどに関する情報を含みます。これらのログファイルは公開 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。

> **Note**
>
> `requestID` は、 から出力されるログの信頼できる一意の ID ではありません AWS AppSync。はクライアントによって上書き`requestID`できます。したがって、この情報に基づいて意思決定を行うときは注意が必要です。

次の例は `CreateApiKey` オペレーションを示す CloudTrail ログエントリです。

```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/Alice",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "diego_ramirez"
    },
    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "CreateApiKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
```

```
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
    "requestParameters": {
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": {
      "apiKey": {
      "id": "***",
      "expires": 1518037200000
      }
    },
    "requestID": "99999999-9999-9999-9999-999999999999",
    "eventID": "99999999-9999-9999-9999-999999999999",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
    }
  ]
}
```

次の例は ListApiKeys オペレーションを示す CloudTrail ログエントリです。

```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/diego_ramirez",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "diego_ramirez"
    },
    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "ListApiKeys",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
    "requestParameters": {
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": {
      "apiKeys": [
```

```
            {
                  "id": "***",
                  "expires": 1517954400000
            },
            {
                  "id": "***",
                  "expires": 1518037200000
            },
          ]
    },
    "requestID": "99999999-9999-9999-9999-999999999999",
    "eventID": "99999999-9999-9999-9999-999999999999",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
    }
  ]
}
```

次の例は DeleteApiKey オペレーションを示す CloudTrail ログエントリです。

```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/diego_ramirez",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "diego_ramirez"
    },
    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "DeleteApiKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
    "requestParameters": {
      "id": "***",
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": null,
```

```
    "requestID": "99999999-9999-9999-9999-999999999999",
    "eventID": "99999999-9999-9999-9999-999999999999",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
    }
  ]
}
```

次の CloudTrail ログエントリの例は、カスタム Lambda 関数オーソライザーで承認された正常な GraphQL ミューテーションを示しています。

```
{
  "eventVersion": "1.10",
    "userIdentity": {
      "type": "Unknown"
    },
    "eventTime": "2024-11-06T15:42:30Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "GraphQL",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "15.248.1.214",
    "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:131.0)
 Gecko/20100101 Firefox/131.0",
    "requestParameters": null,
    "responseElements": null,
    "additionalEventData": {
      "operationName": "MyMutation",
      "authType": [
        "AWS_LAMBDA"
      ],
      "fieldAuthorizationResults": {
        "deniedFields": []
      }
    },
    "requestID": "c2d3768b-3446-40a1-bd95-8399fe776f96",
    "eventID": "21568be1-a1a8-4f43-b978-63cb4cc02a96",
    "readOnly": false,
    "resources": [
    {
      "accountId": "123456789012",
      "type": "AWS::AppSync::GraphQLApi",
      "ARN": "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u"
```

```
    }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": false,
    "recipientAccountId": "123456789012",
    "eventCategory": "Data"
}
```

次の CloudTrail ログエントリの例は、カスタム Lambda 関数オーソライザーで承認された GraphQL オペレーションが部分的に成功したことを示しています。拒否フィールドを指定する `fieldAuthorizationResults.deniedFields`プロパティを書き留めます。

```
{
  "eventVersion": "1.10",
  "userIdentity": {
    "type": "Unknown"
  },
  "eventTime": "2024-11-06T16:11:49Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "GraphQL",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "15.248.1.214",
  "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:131.0) Gecko/20100101
 Firefox/131.0",
  "requestParameters": null,
  "responseElements": null,
  "additionalEventData": {
    "operationName": "MyMutation",
    "authType": [
      "AWS_LAMBDA"
    ],
    "fieldAuthorizationResults": {
      "deniedFields": [
        "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u/types/
Mutation/fields/createPost",
        "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u/types/
Subscription/fields/onCreatePost",
        "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u/types/
Post/fields/status"
      ]
    }
  },
  "requestID": "ae817c4c-66ba-4f64-92a5-ba9c9c341dcd",
```

```
  "eventID": "30109698-7605-476a-9dff-b7ed78d134dc",
  "readOnly": false,
  "resources": [
    {
      "accountId": "123456789012",
      "type": "AWS::AppSync::GraphQLApi",
      "ARN": "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": false,
  "recipientAccountId": "123456789012",
  "eventCategory": "Data"
}
```

次の CloudTrail ログエントリの例は、失敗した GraphQL オペレーションを示しています。

```
{
  "eventVersion": "1.10",
  "userIdentity": {
    "type": "Unknown"
  },
  "eventTime": "2024-11-06T15:51:11Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "GraphQL",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "15.248.1.214",
  "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:131.0) Gecko/20100101
 Firefox/131.0",
  "errorCode": "AccessDenied",
  "errorMessage": "{\n \"errors\" : [ {\n \"errorType\" : \"UnauthorizedException\",\n
 \"message\" : \"You are not authorized to make this call.\"\n } ]\n}",
  "requestParameters": null,
  "responseElements": null,
  "additionalEventData": {
    "operationName": "MyFullyDeniedLambdaMutation"
  },
  "requestID": "0bef3cf3-a48b-4de9-8b1f-038afb563516",
  "eventID": "b738651f-4ec0-4548-8fec-200c6b42842b",
  "readOnly": false,
  "resources": [
    {
      "accountId": "123456789012",
```

```
      "type": "AWS::AppSync::GraphQLApi",
      "ARN": "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": false,
  "recipientAccountId": "123456789012",
  "eventCategory": "Data"
}
```

次の例は、GraphQL リクエストの成功を示しています。

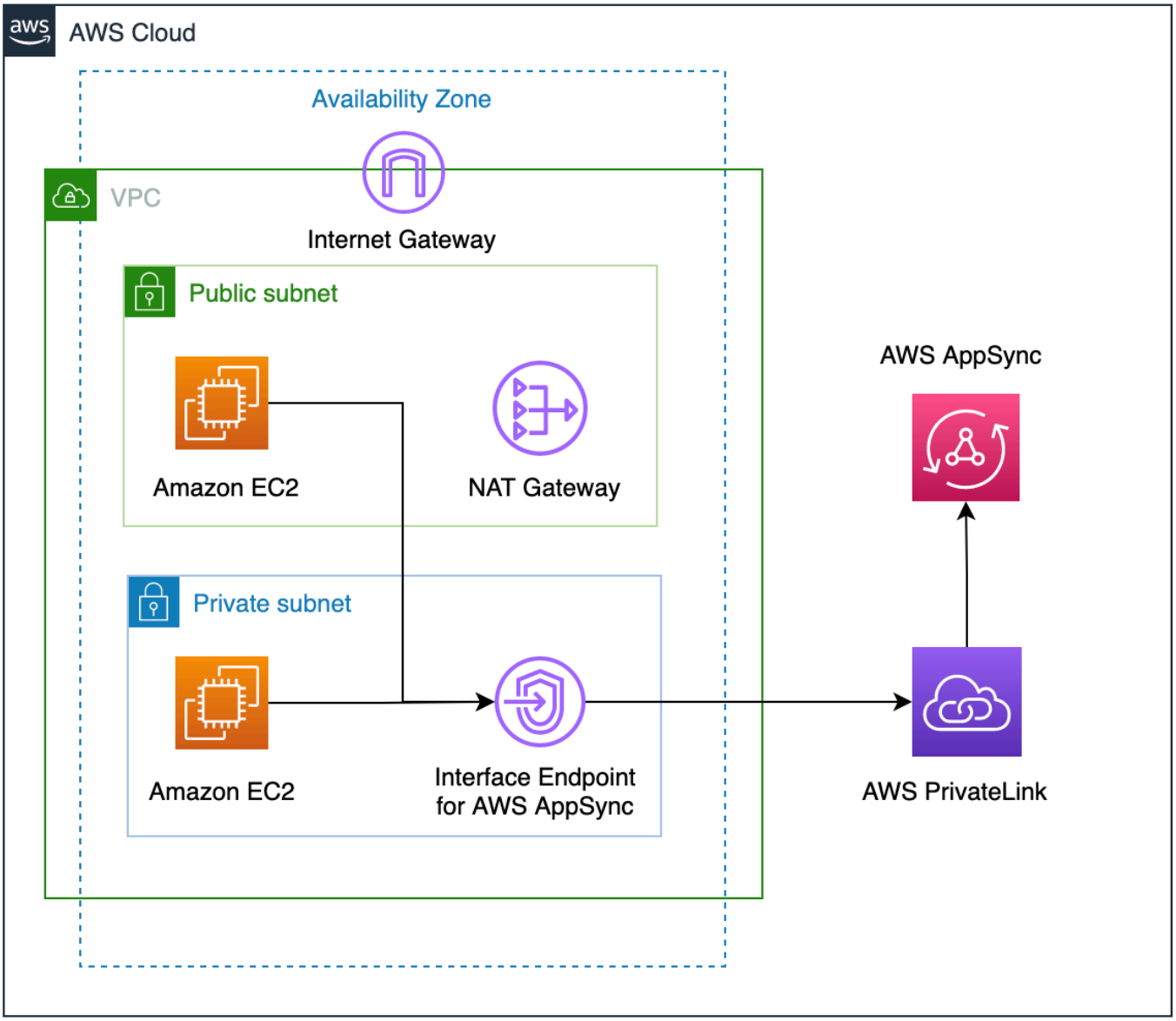
```
{
  "eventVersion": "1.10",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AIDACKCEVSQ6C2EXAMPLE:jane_doe",
    "arn": "arn:aws:sts::123456789012:assumed-role/admin/jane_doe",
    "accountId": "123456789012",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/admin",
        "accountId": "123456789012",
        "userName": "jane_doe"
      },
      "attributes": {
        "creationDate": "2024-11-06T15:40:09Z",
        "mfaAuthenticated": "false"
      }
    }
  },
  "eventTime": "2024-11-06T16:03:43Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "GraphQL",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "15.248.1.214",
  "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:131.0) Gecko/20100101
 Firefox/131.0",
  "requestParameters": null,
  "responseElements": null,
  "additionalEventData": {
```

```
    "operationName": "IamFullSuccess",
    "authType": [
      "AWS_IAM"
    ],
    "fieldAuthorizationResults": {
      "allowedFields": [
        "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u/types/
Mutation/fields/createSecondPostAllowed"
      ],
      "deniedFields": []
    }
  },
  "requestID": "edc6bbbf-6bf2-40f5-820f-ef444f12e0c1",
  "eventID": "524656a5-0925-4370-9e7e-08888e9c299f",
  "readOnly": false,
  "resources": [
    {
      "accountId": "123456789012",
      "type": "AWS::AppSync::GraphQLApi",
      "ARN": "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": false,
  "recipientAccountId": "123456789012",
  "eventCategory": "Data"
}
```


AWS AppSync プライベート APIs

Amazon Virtual Private Cloud (Amazon VPC) を使用する場合は、 AWS AppSync プライベート APIs を作成できます。これは、VPC からのみアクセスできる APIsです。プライベート API を使用すると、データを公開することなく、内部アプリケーションへの API アクセスを制限し、GraphQL および Realtime エンドポイントに接続できます。

VPC と AWS AppSync サービス間のプライベート接続を確立するには、インターフェイス VPC エンドポイントを作成する必要があります。インターフェイスエンドポイントは を利用しておりAWS PrivateLink、インターネットゲートウェイ、NAT デバイス、VPN 接続、または AWS Direct Connect 接続なしで AWS AppSync APIs にプライベートにアクセスできます。VPC のインスタンスは、パブリック IP アドレスがなくても AWS AppSync API と通信できます。VPC と の間のトラフィック AWS AppSync は、 AWS ネットワークを離れません。



プライベート API 機能を有効にする前に考慮すべき要素は他にもいくつかあります。

- プライベート DNS 機能を有効に AWS AppSync して の VPC インターフェイスエンドポイントを設定すると、VPC 内のリソースが AWS AppSync 生成された APIs URL を使用して他の AWS AppSync パブリック API を呼び出すことができなくなります。これは、パブリック API へのリクエストがインターフェイスエンドポイント経由でルーティングされるためで、パブリック API では許可されていません。このシナリオでパブリック API を呼び出すには、パブリック API にカスタムドメイン名を設定し、VPC 内のリソースがそのドメイン名を使用してパブリック API を呼び出すことをお勧めします。

- プライベート APIs AWS AppSync は VPC からのみ使用できます。 AWS AppSync コンソールクエリエディタは、ブラウザのネットワーク設定が VPC にトラフィックをルーティングできる (VPN 経由の接続など AWS Direct Connect) 場合にのみ API に到達できます。
- の VPC インターフェイスエンドポイントを使用すると AWS AppSync、同じ AWS アカウントとリージョン内の任意のプライベート API にアクセスできます。プライベート API へのアクセスをさらに制限するには、以下のオプションを検討してください。
  - 必要な管理者のみが VPC エンドポイントインターフェイスを作成できるようにします AWS AppSync。
  - VPC エンドポイントのカスタムポリシーを使用して、VPC 内のリソースから呼び出せる API を制限します。
  - VPC 内のリソースについては、IAM 認可を使用して AWS AppSync API を呼び出すことをお勧めします。そのためには、リソースに API に対してスコープダウンされたロールが割り当てられていることを確認する必要があります。
- IAM プリンシパルを制限するポリシーを作成または使用するときは、メソッドの `authorizationType` を `AWS_IAM` または `NONE` に設定する必要があります。

AWS AppSync プライベート APIsの作成

次の手順は、 AWS AppSync サービスでプライベート APIsを作成する方法を示しています。

> **Warning**
> プライベート API 機能は API の作成中にのみ有効にできます。この設定は、API AWS AppSync または AWS AppSync プライベート API の作成後に変更することはできません。

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   - ダッシュボードで、[API の作成] を選択します。
2. [API を最初から設計する] を選択し、[次へ] を選択します。
3. [プライベート API] セクションで、[プライベート API 機能を使用する] を選択します。
4. 残りのオプションを設定し、API のデータを確認して、[作成] を選択します。

AWS AppSync Private API を使用する前に、VPC AWS AppSync で のインターフェイスエンドポイントを設定する必要があります。Private API と VPC の両方が同じ AWS アカウントとリージョンに存在する必要があることに注意してください。

AWS AppSyncのインターフェイスエンドポイントの作成

Amazon VPC コンソールまたは AWS Command Line Interface () AWS AppSync を使用して、 のインターフェイスエンドポイントを作成できますAWS CLI。詳細については、 Amazon VPC ユーザーガイド のインターフェイスエンドポイントの作成を参照してください。

Console

1. にサインイン AWS Management Console し、Amazon VPC コンソールのエンドポイントページを開きます。
2. [エンドポイントの作成] を選択します。
   a. [サービスカテゴリ] フィールドで、[AWS のサービス] が選択されていることを確認します。
   b. [サービス] テーブルで、[com.amazonaws.{region}.appsync-api] を選択します。Type 列の値が Interface であることを確認します。
   c. VPC フィールドで、VPC とそのサブネットを選択します。
   d. インターフェイスエンドポイントのプライベート DNS を有効にするには、[DNS 名を有効にする] チェックボックスをオンにします。
   e. [セキュリティグループ] フィールドで、1 つ以上のセキュリティグループを選択します。
3. [エンドポイントの作成] を選択します。

CLI

create-vpc-endpoint コマンドを使用し、VPC ID、VPC エンドポイントタイプ (インターフェイス)、サービス名、エンドポイントを使用するサブネット、およびエンドポイントネットワークインターフェイスに関連付けるセキュリティグループを指定します。以下に例を示します。

```
$ aws ec2 create-vpc-endpoint –vpc-id vpc-ec43eb89 \
  –vpc-endpoint-type Interface \
  –service-name com.amazonaws.{region}.appsync-api \
```

```
  –subnet-id subnet-abababab –security-group-id sg-1a2b3c4d
```

プライベート DNS オプションを使用するには、VPC の enableDnsHostnames および enableDnsSupportattributes を設定する必要があります。詳細については、「Amazon VPC ユーザーガイド」の「VPC の DNS 属性の表示と更新」を参照してください。インターフェイスエンドポイントのプライベート DNS 機能を有効にすると、以下の形式を使用して、デフォルトのパブリック DNS エンドポイントを使用して AWS AppSync API GraphQL およびリアルタイムエンドポイントにリクエストを行うことができます。

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

詳細については、アマゾン ウェブ サービス全般のリファレンスの「サービスエンドポイントとクォータ」を参照してください。

詳細については、Amazon VPC ユーザーガイドの「インターフェイスエンドポイントを介したサービスへのアクセス」を参照してください。

AWS CloudFormation を使用してエンドポイントを作成および設定する方法については、AWS CloudFormation ユーザーガイドの AWS::EC2::VPCEndpoint リソースを参照してください。

高度な の例

インターフェイスエンドポイントのプライベート DNS 機能を有効にすると、以下の形式を使用して、デフォルトのパブリック DNS エンドポイントを使用して AWS AppSync API GraphQL およびリアルタイムエンドポイントにリクエストを行うことができます。

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

インターフェイスの VPC エンドポイントのパブリック DNS ホスト名を使用すると、API を呼び出すためのベース URL は次の形式になります。

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.
{region}.vpce.amazonaws.com/graphql
```

AZ にエンドポイントをデプロイしている場合は、AZ 固有の DNS ホスト名を使用することもできます。

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}-{az_id}.appsync-api.
{region}.vpce.amazonaws.com/graphql.
```

VPC エンドポイントのパブリック DNS 名を使用するには、 AWS AppSync API エンドポイントのホスト名を Hostまたは としてリクエストの x-appsync-domainヘッダーとして渡す必要があります。以下の例では、サンプルスキーマの起動TodoAPIガイドで作成されたものを使用しています。

```
curl https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.
{region}.vpce.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-H "Host:{api_url_identifier}.appsync-api.{region}.amazonaws.com" \
-d '{"query":"mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input:
 $createtodoinput) {\n id\n name\n where\n when\n description\n }\n}","variables":
{"createtodoinput":{"name":"My first GraphQL task","when":"Friday Night","where":"Day
 1","description":"Learn more about GraphQL"}}}'
```

以下の例では、サンプルスキーマの起動ガイドで生成された Todo アプリを使用します。サンプル Todo API をテストするために、プライベート DNS を使用して API を呼び出します。任意のコマンドラインツールを使用できます。この例では curl を使用してクエリとミューテーションを送信し、wscat を使用してサブスクリプションを設定します。この例をエミュレートするには、以下のコマンド{ }の角括弧内の値を AWS 、アカウントからの対応する値に置き換えます。

ミューテーション操作のテスト — **createTodo** リクエスト

```
curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query":"mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input:
 $createtodoinput) {\n id\n name\n where\n when\n description\n }\n}","variables":
{"createtodoinput":{"name":"My first GraphQL task","when":"Friday Night","where":"Day
 1","description":"Learn more about GraphQL"}}}'
```

ミューテーション操作のテスト — **createTodo** 応答

```
{
    "data": {
        "createTodo": {
            "id": "<todo-id>",
            "name": "My first GraphQL task",
```

```
            "where": "Day 1",
            "when": "Friday Night",
            "description": "Learn more about GraphQL"
        }
    }
}
```

クエリ操作のテスト — **listTodos** リクエスト

```
curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query":"query ListTodos {\n listTodos {\n items {\n description\n id\n name
\n when\n where\n }\n }\n}\n","variables":{"createtodoinput":{"name":"My first
 GraphQL task","when":"Friday Night","where":"Day 1","description":"Learn more about
 GraphQL"}}}'
```

クエリ操作のテスト — **listTodos** リクエスト

```
{
  "data": {
    "listTodos": {
      "items": [
        {
          "description": "Learn more about GraphQL",
          "id": "<todo-id>",
          "name": "My first GraphQL task",
          "when": "Friday night",
          "where": "Day 1"
        }
      ]
    }
  }
}
```

サブスクリプション操作のテスト — **createTodo** ミューテーションへのサブスクライブ

で GraphQL サブスクリプションを設定するには AWS AppSync、「リアルタイム WebSocket クライアントの構築」を参照してください。VPC の Amazon EC2 インスタンスから、wscat を使用して AWS AppSync プライベート API サブスクリプションエンドポイントをテストできます。以下の例では、認可に API KEY を使用しています。

```
$ header=`echo '{"host":"{api_url_identifier}.appsync-api.{region}.amazonaws.com","x-
api-key":"da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}"}' | base64 | tr -d '\n'`
$ wscat -p 13 -s graphql-ws -c  "wss://{api_url_identifier}.appsync-realtime-api.us-
west-2.amazonaws.com/graphql?header=$header&payload=e30="
Connected (press CTRL+C to quit)
> {"type": "connection_init"}
< {"type":"connection_ack","payload":{"connectionTimeoutMs":300000}}
< {"type":"ka"}
> {"id":"f7a49717","payload":{"data":"{\"query\":\"subscription
 onCreateTodo {onCreateTodo {description id name where when}}\",
\"variables\":{}}","extensions":{"authorization":{"x-api-key":"da2-
{xxxxxxxxxxxxxxxxxxxxxxxxxx}","host":"{api_url_identifier}.appsync-api.
{region}.amazonaws.com"}}},"type":"start"}
< {"id":"f7a49717","type":"start_ack"}
```

または、VPC エンドポイントのドメイン名を使用し、wscatウェブソケットを確立するコマンドで必ず Host ヘッダーを指定してください。

```
$ header=`echo '{"host":"{api_url_identifier}.appsync-api.{region}.amazonaws.com","x-
api-key":"da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}"}' | base64 | tr -d '\n'`
$ wscat -p 13 -s graphql-ws -c  "wss://{vpc_endpoint_id}-
{endpoint_dns_identifier}.appsync-api.{region}.vpce.amazonaws.com/graphql?header=
$header&payload=e30=" --header Host:{api_url_identifier}.appsync-realtime-api.us-
west-2.amazonaws.com
Connected (press CTRL+C to quit)
> {"type": "connection_init"}
< {"type":"connection_ack","payload":{"connectionTimeoutMs":300000}}
< {"type":"ka"}
> {"id":"f7a49717","payload":{"data":"{\"query\":\"subscription
 onCreateTodo {onCreateTodo {description id priority title}}\",
\"variables\":{}}","extensions":{"authorization":{"x-api-key":"da2-
{xxxxxxxxxxxxxxxxxxxxxxxxxx}","host":"{api_url_identifier}.appsync-api.
{region}.amazonaws.com"}}},"type":"start"}
< {"id":"f7a49717","type":"start_ack"}
```

以下のミューテーションコードを実行します。

```
curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query":"mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input:
 $createtodoinput) {\n id\n name\n where\n when\n description\n }\n}","variables":
```

```
{"createtodoinput":{"name":"My first GraphQL task","when":"Friday Night","where":"Day
 1","description":"Learn more about GraphQL"}}}'
```

その後、サブスクリプションがトリガーされ、次のようなメッセージ通知が表示されます。

```
< {"id":"f7a49717","type":"data","payload":{"data":{"onCreateTodo":{"description":"Go
 to the shops","id":"169ce516-b7e8-4a6a-88c1-ab840184359f","priority":5,"title":"Go to
 the shops"}}}}
```

IAM ポリシーを使用してパブリック API の作成を制限する

AWS AppSync は、プライベート APIs で使用する IAM Conditionステートメントをサポートしています。visibilityフィールドを appsync:CreateGraphqlApi オペレーションの IAM ポリシーステートメントに含めて、どの IAM ロールとユーザーがプライベート API とパブリック API を作成できるかを制御できます。これにより、IAM 管理者は、ユーザーにプライベート GraphQL API の作成のみを許可する IAM ポリシーを定義できます。ユーザーがパブリック API を作成しようとすると、許可されていないメッセージが届きます。

たとえば、IAM 管理者はプライベート API の作成を許可する次の IAM ポリシーステートメントを作成できます。

```
{
    "Sid": "AllowPrivateAppSyncApis",
    "Effect": "Allow",
    "Action": "appsync:CreateGraphqlApi",
    "Resource": "*",
    "Condition": {
        "ForAnyValue:StringEquals": {
            "appsync:Visibility": "PRIVATE"
        }
    }
}
```

IAM 管理者は、次のサービスコントロールポリシーを追加して、 AWS 組織内のすべてのユーザーがプライベート AWS AppSync APIs以外の API を作成するAPIsブロックすることもできます。

```
{
    "Sid": "BlockNonPrivateAppSyncApis",
    "Effect": "Deny",
    "Action": "appsync:CreateGraphqlApi",
```

```
    "Resource": "*",
    "Condition": {
        "ForAnyValue:StringNotEquals": {
            "appsync:Visibility": "PRIVATE"
        }
    }
}
```

Sharing AWS AppSync GraphQL APIs

AWS AppSync は AWS Resource Access Manager（ AWS RAM) と統合してリソース共有を有効にします。 AWS RAM は、 AWS AppSync GraphQL APIs の呼び出しアクション (クエリ、ミューテーション、サブスクリプションオペレーション、リアルタイム WebSocket エンドポイントへのリクエストの接続) を他の AWS アカウント または を通じて共有できるサービスです AWS Organizations。では AWS RAM、リソース共有を作成して、所有しているリソースを共有します。リソース共有は、共有するリソースと、それらを共有するコンシューマーを指定します。コンシューマーには、以下を含めることができます。

- の組織 AWS アカウント 内外に固有 AWS Organizations
- の組織内の組織単位 AWS Organizations
- の組織全体 AWS Organizations

詳細については AWS RAM、AWS Resource Access Manager「 ユーザーガイド」を参照してください。

トピック

- AWS AppSync GraphQL APIs を共有するための前提条件
- Share AWS AppSync GraphQL APIs
- AWS AppSync GraphQL APIs
- クロスアカウントイベント

AWS AppSync GraphQL APIs を共有するための前提条件

Sharing AWS AppSync GraphQL APIs以下の前提条件があります。

- AWS AppSync GraphQL API を共有するには、 でその API を所有している必要があります AWS アカウント。つまり、 AWS AppSync GraphQL API はアカウントで割り当てまたはプロビジョニングする必要があります。
- の組織または組織単位と AWS AppSync GraphQL API を共有するには AWS Organizations、 との共有を有効にする必要があります AWS Organizations。詳細については、「AWS Resource Access Manager ユーザーガイド」の「AWS Organizations内でリソース共有を有効にする」を参照してください。

Share AWS AppSync GraphQL APIs

AWS AppSync GraphQL API を共有するには、まず を使用してリソース共有を作成します AWS Resource Access Manager。リソース共有では、共有対象のリソース、リソースを共有するコンシューマー、プリンシパルが実行できるアクションを指定します。所有している AWS AppSync GraphQL API を他の と共有 AWS アカウントする場合、それらのアカウントが でその AWS AppSync API を呼び出すことを有効にします AWS アカウント。

ユーザーが の組織のメンバーであり AWS Organizations、組織内での共有が有効になっている場合、組織内のコンシューマーには共有リソースへのアクセスが自動的に付与されます。これに該当しない場合、コンシューマーはリソースへの参加の招待を受け取り、その招待を受け入れた後で、リソースの共有に対するアクセス許可が付与されます。

共有に関する考慮事項

- 共有できるのは AWS AppSync GraphQL APIs のみです。Event API などの他の APIs タイプは共有できません。
- API で設定された認可モードの 1 つ`AWS_IAM`として APIs を持つ AWS AppSync GraphQL API のみを共有できます。

  `AWS_IAM` が共有 AppSync GraphQL API の認可モードのリストから削除され、リソース共有がまだ存在する場合でも、無効になります。
- public と private AWS AppSync GraphQL APIsの両方を共有できます。
- Private AWS AppSync GraphQL APIs は、オリジンの VPC の VPCs エンドポイントを介して常にアクセスでき AWS アカウント、 だけでなく、すべての認可モードがサポートされていますAWS_IAM。
- shared AWS AppSync GraphQL APIs の場合、アクセス許可は API リソースに対してのみ管理され、フィールドとタイプ、およびフィールドリソースに対するきめ細かなアクセス許可はサポート

されていません。API を共有すると、API ARN とそのすべてのタイプとフィールドARNs を共有します。

AWS RAM コンソールを使用して、所有するリソース共有を作成する

AWS AppSync GraphQL API を共有するには、AWS Resource Access Manager 「 ユーザーガイド」の「アクセスRAM許可名 を使用してリソース共有を作成する」で説明されている手順を使用しますAWSRAMPermissionAppSyncGraphQLApiInvokeAccess。

AWS RAM コンソールを使用して、カスタマー管理アクセス許可を作成して使用して private AWS AppSync GraphQL API を共有する

private AWS AppSync GraphQL API を共有するには、AWS Resource Access Manager 「 ユーザーガイド」の「カスタマー管理アクセス許可の作成と使用」で説明されている手順を使用して、カスタマー管理アクセス許可を作成します。

例えば、アカウント A の所有者は、VPCE-B (アカウント B が所有する VPC エンドポイント) を介して行われた呼び出しに対して、アカウント B のプリンシパルに private AWS AppSync GraphQL API (PrivateApiA) にアクセスする許可を付与したいと考えています。この場合、アカウント A の所有者は、次のように AWS RAM カスタマー管理アクセス許可を作成する必要があります。

```
{
    "Effect": "Allow",
    "Action": ["appsync:GraphQL"],
    "Condition": {
        "StringEqualsIgnoreCase": {
            "aws:SourceVpce": [
                "VPCE-B"
            ]
        }
    }
}
```

この新しいカスタマー管理アクセス AWS RAM 許可の名前が であるとしますprivate-api-A-access-via-vpce-b。

PrivateApiA 経由で へのクロスアカウントアクセスを有効にするにはVPCE-B、次のパラメータと前の例のカスタマー管理アクセス許可を使用して AWS RAM リソース共有を作成できます。

- リソースタイプ： appsync:Apis

- リソース: `arn:aws:appsync:us-west-2:A:apis/PrivateApiA`
- アクセス許可: `private-api-A-access-via-vpce-b` (カスタマー管理アクセス許可)
- プリンシパル : `Account: B`

を使用して、所有するリソース共有を作成する AWS CLI

を使用して AWS AppSync GraphQL API を共有するには AWS CLI、 `--permission-arns` スイッチの値`arn:aws:ram::aws:permission/AWSRAMPermissionAppSyncApiInvokeAccess`として を使用する `create-resource-share` コマンドを使用します。

で使用できるコマンドの完全なリストについては AWS RAM、 AWS RAM CLI リファレンスを参照してください。

AWS AppSync GraphQL APIs

所有している AWS AppSync GraphQL APIsの共有を停止するには、リソース共有を削除するか、リソースを共有したプリンシパルを更新する必要があります。実行するアクションについては、以下のセクションのドキュメントを参照してください。

AWS RAM コンソールを使用して所有しているリソースの共有を停止するには

「AWS Resource Access Manager ユーザーガイド」の「リソース共有の更新」を参照してください。

を使用して所有しているリソースの共有を停止するには AWS CLI

disassociate-resource-share コマンドを使用します。

AWS RAM コンソールを使用して所有しているリソース共有を削除するには

AWS Resource Access Manager 「 ユーザーガイド」の「リソース共有の削除」を参照してください。

を使用して所有しているリソース共有を削除するには AWS CLI

delete-resource-share コマンドを使用します。

で使用できるコマンドの完全なリストについては AWS RAM、 AWS RAM CLI リファレンスを参照してください。

クロスアカウントイベント

クロスアカウント AWS AppSync GraphQL API AWS CloudTrail DataPlane アクティビティのモニタリングと監査のために、データイベントのログ記録をオプトインできます。詳細については、「AWS CloudTrail ユーザーガイド」の「データイベントをログ記録する」を参照してください。

GraphQL の実行の複雑さ、クエリの深さ、イントロスペクションを AWS AppSyncで設定する

AWS AppSync では、イントロスペクション機能を有効または無効にし、1 つのクエリでネストされたレベルとリゾルバーの量に制限を設定できます。

イントロスペクション機能を使用する

> **Tip**
>
> GraphQL のイントロスペクションの詳細については、GraphQL Foundation のウェブサイトにあるこちらの記事を参照してください。

GraphQL では、デフォルトでイントロスペクションを使用してスキーマ自体にクエリを実行し、その型、フィールド、クエリ、ミューテーション、サブスクリプションなどを検出できます。これは、GraphQL サービスによってデータがどのように形成され、処理されるかを学習するための重要な機能です。ただし、イントロスペクションを使用する際には、いくつか考慮すべき点があります。例えば、フィールド名が機密または非表示である場合や、API スキーマ全体をコンシューマー向けに文書化しないでおくことが意図されている場合など、イントロスペクションを無効にすることで有利となるユースケースがあるかもしれません。このような場合、イントロスペクションを通じてスキーマデータを公開すると、意図的にプライベートデータが漏洩する可能性があります。

そのような事態を防ぐために、イントロスペクションを無効にすることができます。これにより、権限のない者がスキーマのイントロスペクションフィールドを使用するのを防ぐことができます。ただし、イントロスペクションは、開発チームにとってサービス内のデータがどのように処理されるかを知るのに役立つことに注意してください。内部的には、セキュリティーを強化するために本番稼働用コードでイントロスペクションを無効にしつつ、イントロスペクションを有効にしておくと役立つ場合があります。もう 1 つの対処方法は、認可方法を追加することです。これは AWS AppSync でも提供されています。詳細については、「認可」を参照してください。

AWS AppSync では、API レベルでイントロスペクションを有効または無効にできます。イントロスペクションを有効または無効にするには、次の手順を実行します。

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
2. API] ページで、GraphQL API の名前を選択します。
3. API のホームページのナビゲーションペインで、[設定] を選択します。
4. [API 設定] で [編集] を選択します。
5. [内観クエリ] で、以下の手順を実行します。
   - [内観クエリを有効化] をオンまたはオフにします。
6. [保存] を選択します。

イントロスペクションが有効 (デフォルトの動作) になっている場合、イントロスペクションシステムの使用は正常に機能します。例えば、以下の画像は、スキーマ内で使用可能なすべての型を処理する __schema フィールドを示しています。

この機能を無効にすると、代わりに検証エラーがレスポンスに表示されます。

クエリの深さの制限を設定する

操作中に API の機能をよりきめ細かく制御したい場合もあるでしょう。そのような制御の 1 つとして、クエリが処理できるネストレベルの数に制限を追加することが挙げられます。デフォルトでは、クエリはネストレベルを無制限に処理できます。クエリを指定された数のネストレベルに制限すると、プロジェクトのパフォーマンスと柔軟性に影響が出る可能性があります。次のようなクエリがあるとします。

```
query MyQuery {
  L1: nextLayer {
    L2: nextLayer {
      L3: nextLayer {
        L4: value
      }
    }
  }
}
```

プロジェクトでは、何らかの目的でクエリを L1 または L2 に制限することが求められる場合があります。デフォルトでは、L1 から L4 までのクエリ全体が処理され、それを制御する方法はありません。制限を設定することで、指定したレベルを超えるものにクエリがアクセスするのを防ぐことができます。

クエリの深さの制限を追加するには、次の手順を実行します。

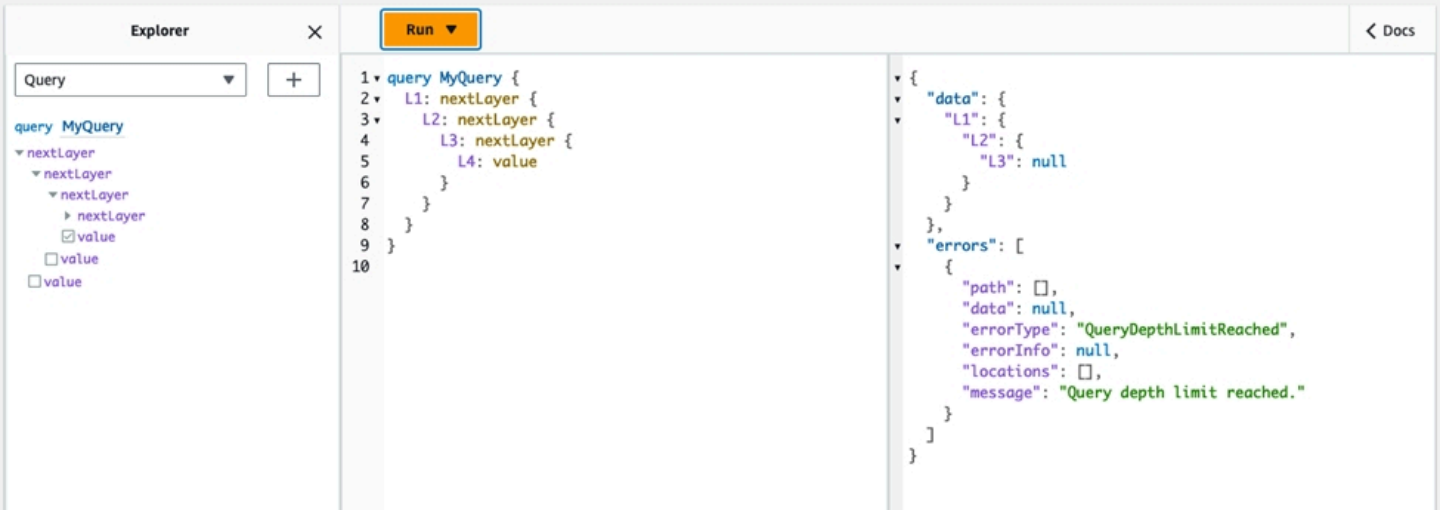
1. にサインイン AWS Management Console し、AppSync コンソールを開きます。

2. API] ページで、GraphQL API の名前を選択します。
3. API のホームページのナビゲーションペインで、[設定] を選択します。
4. [API 設定] で [編集] を選択します。
5. [クエリーの深さ] で、以下の操作を行います。

   a. [クエリーの深さを有効化] をオンまたはオフにします。

   b. [最大深度] で、深度の制限を設定します。これは 1 と 75 の範囲で指定できます。
6. [Save] を選択します。

制限が設定されている場合、その上限を超えると QueryDepthLimitReached エラーが発生します。例えば、以下の画像は、深さ制限が 2 のクエリが制限を超えて 3 番目 (L3) と 4 番目 (L4) のレベルに達していることを示しています。



ただし、スキーマではフィールドを NULL 許容または NULL 非許容としてマークできることに注意してください。NULL 非許容フィールドが QueryDepthLimitReached エラーを受け取った場合、そのエラーは最初の NULL 許容の親フィールドにスローされます。

リゾルバー数の制限の設定

各クエリが処理できるリゾルバーの数を制御することもできます。クエリの深さと同様に、この量にも制限を設定することが可能です。3 つのリゾルバーを含む次のクエリを考えてみましょう。

```
query MyQuery {
  resolver1: resolver
  resolver2: resolver
  resolver3: resolver
```

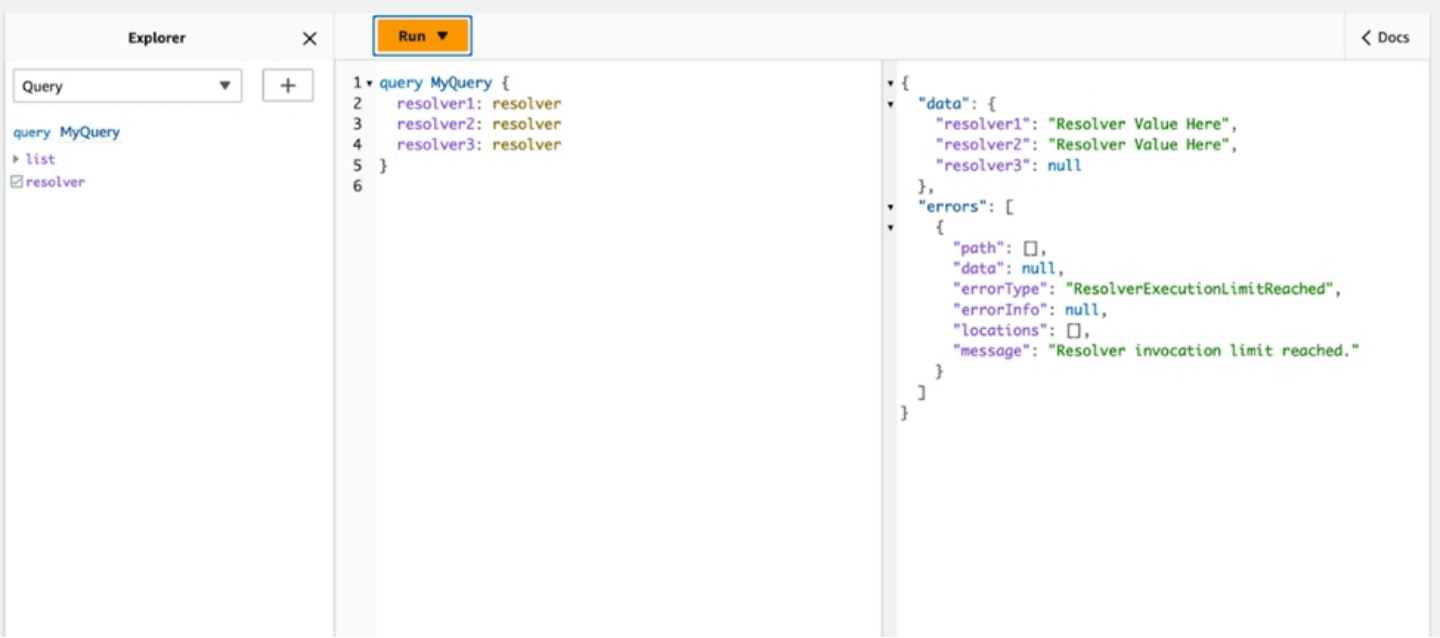
```
}
```

デフォルトでは、各クエリは最大 10,000 個のリゾルバーを処理できます。上の例では、`resolver1`、`resolver2`、`resolver3` が処理されます。ただし、プロジェクトでは、各クエリを合計 1 つまたは 2 つのリゾルバーの処理に制限するように求められる場合があります。制限を設定することで、1 つ目のリゾルバー (`resolver1`) や 2 つ目のリゾルバー (`resolver2`) など、特定の数を超えるリゾルバーを処理しないようにクエリに指示できます。

リゾルバー数の制限を追加するには、次の手順を実行します。

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
2. API] ページで、GraphQL API の名前を選択します。
3. API のホームページのナビゲーションペインで、[設定] を選択します。
4. [API 設定] で [編集] を選択します。
5. [リゾルバー数の制限] で、以下の操作を行います。
   a. [リゾルバー数を有効化] をオンにします。
   b. [最大リゾルバー数] で、数の制限を設定します。これは `1` と `10000` の範囲で指定できます。
6. [Save] を選択します。

クエリの深さの制限と同様、設定されてたリゾルバーの制限を超えると、他のリゾルバーでクエリが `ResolverExecutionLimitReached` エラーで終了します。以下の画像では、リゾルバー数の制限が 2 のクエリが 3 つのリゾルバーを処理しようとします。制限のため、3 番目のリゾルバーはエラーをスローし、実行されません。



での環境変数の使用 AWS AppSync

環境変数を使用して、コードを更新せずに AWS AppSync リゾルバーと関数の動作を調整できます。環境変数は、実行時にリゾルバーと関数が活用できる、API 設定に保存されている文字列のペアです。これらは、初期設定時にのみ使用できる設定データを参照する必要がありますが、実行時にリゾルバーと関数で使用する必要がある状況で特に便利です。環境変数はコード内の設定データを公開するため、これらの値をハードコードする必要性が軽減されます。

> **Note**
>
> データベースのセキュリティを強化するには、環境変数の代わりに Secrets Manager または AWS Systems Manager Parameter Store を使用して認証情報や機密情報を保存することをお勧めします。この機能を活用するには、AWS AppSync 「HTTP データソースを使用した AWS サービスの呼び出し」を参照してください。

環境変数が適切に機能するには、いくつかの動作とルールに従う必要があります。

- JavaScript リゾルバー/関数と VTL テンプレートの両方が環境変数をサポートしています。
- 環境変数は、関数の呼び出しの前には評価されません。
- 環境変数は文字列値のみをサポートします。
- 環境変数で定義した値は文字列リテラルとみなされ、展開されません。

- 変数評価は、理想的には関数コードで実行する必要があります。

環境変数の設定 (コンソール)

AWS AppSync GraphQL API の環境変数を設定するには、変数を作成し、そのキーと値のペアを定義します。リゾルバーと関数は、環境変数のキー名を使用して、実行時に値を取得します。 AWS AppSync コンソールで環境変数を設定するには：

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
2. API] ページで、GraphQL API の名前を選択します。
3. API のホームページのナビゲーションペインで、[設定] を選択します。
4. [環境変数] の下で、[環境変数を追加] を選択します。
5. [環境変数の追加] を選択します。
6. キーと値を入力します。
7. 必要に応じて、ステップ 5 と 6 を繰り返してキー値を追加します。キー値を削除する必要がある場合は、[削除] オプションと削除するキー (複数可) を選択します。
8. [送信] を選択します。

> **Tip**
>
> キーと値を作成するときは、いくつかのルールに従う必要があります。
>
> - キーの先頭は文字を使用する必要があります。
> - キーは 2 文字以上にする必要があります。
> - キーには、文字、数字、およびアンダースコア (_) のみを含めることができます。
> - 値には最大 512 文字を使用できます。
> - GraphQL API では、最大 50 個のキーと値のペアを設定できます。

環境変数の設定 (API)

API を使用して環境変数を設定するには、`PutGraphqlApiEnvironmentVariables` を使用できます。対応する CLI コマンドは `put-graphql-api-environment-variables` です。

API を使用して環境変数を取得するには、GetGraphqlApiEnvironmentVariables を使用できます。対応する CLI コマンドは get-graphql-api-environment-variables です。

コマンドには、API ID と環境変数のリストが含まれている必要があります。

```
aws appsync put-graphql-api-environment-variables \
  --api-id "<api-id>" \
  --environment-variables '{"key1":"value1","key2":"value2", …}'
```

次の例では、put-graphql-api-environment-variables コマンドを使用する ID abcdefghijklmnopqrstuvwxyz を持つ API に 2 つの環境変数を設定します。

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true"}'
```

put-graphql-api-environment-variables コマンドで環境変数を適用すると、環境変数の構造の内容が上書きされることに注意してください。これは、既存の環境変数が失われることを意味します。新しい環境変数を追加するときに、既存の環境変数を保持するには、リクエストに新しい環境変数と一緒に、すべての既存のキーと値のペアを含めます。上記の例を使用して、"EMPTY":"" を追加する場合は、以下を実行できます。

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true", "EMPTY":""}'
```

現在の設定を取得するには、get-graphql-api-environment-variables コマンドを使用します。

```
aws appsync get-graphql-api-environment-variables --api-id "<api-id>"
```

上記の例を使用して、次のコマンドを使用できます。

```
aws appsync get-graphql-api-environment-variables --api-id "abcdefghijklmnopqrstuvwxyz"
```

結果には、環境変数のリストとそのキー値が表示されます。

```
{
    "environmentVariables": {
```

```
        "USER_TABLE": "users_prod",
        "DEBUG": "true",
        "EMPTY": ""
    }
}
```

環境変数の設定 (CFN)

以下のテンプレートを使用して環境変数を作成できます。

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  GraphQLApiWithEnvVariables:
    Type: "AWS::AppSync::GraphQLApi"
    Properties:
      Name: "MyApiWithEnvVars"
      AuthenticationType: "AWS_IAM"
      EnvironmentVariables:
        EnvKey1: "non-empty"
        EnvKey2: ""
```

環境変数とマージ済みの API

ソース API で定義されている環境変数は、マージ済みの API でも利用できます。マージ済みの API 環境変数は読み取り専用であり、更新できません。マージを成功させるには、環境変数キーがすべてのソース API で一意である必要があります。キーが重複すると、マージは常に失敗します。

環境変数の取得

関数コードの環境変数を取得するには、リゾルバーと関数の `ctx.env` オブジェクトから値を取得します。以下に、実際に実行されている例をいくつか示します。

Publishing to Amazon SNS

この例では、HTTP リゾルバーは Amazon SNS トピックにメッセージを送信します。トピックの ARN は、GraphQL API とトピックを定義するスタックがデプロイされた後にのみ認識されます。

```
/**
 * Sends a publish request to the SNS topic
 */
```

```
export function request(ctx) {
    const TOPIC_ARN = ctx.env.TOPIC_ARN;
    const { input: values } = ctx.args;
    // this custom function sends values to the SNS topic
    return publishToSNSRequest(TOPIC_ARN, values);
}
```

Transactions with DynamoDB

この例では、API がステージング用にデプロイされている場合、または既に本番環境にある場合、DynamoDB テーブルの名前は異なります。リゾルバーコードを変更する必要はありません。環境変数の値は、API がデプロイされる場所に基づいて更新されます。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: ctx.env.POST_TABLE,
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({ postId }),
        // rest of the configuration
      },
      {
        table: ctx.env.AUTHOR_TABLE,
        operation: 'UpdateItem',
        key: util.dynamodb.toMapValues({ authorId }),
        // rest of the configuration
      },
    ],
  };
}
```

GraphQL API を保護するための認可と認証の設定

AWS AppSync は、GraphQL APIs を保護するため、API キー、Lambda、IAM、OpenID Connect、および Cognito ユーザープールの認可タイプを提供します。各オプションは、異なるセキュリティ方法を提供します。

1. API キー認可: 認証されていない API のスロットリングを制御し、シンプルなセキュリティオプションを提供します。
2. Lambda 認可: 関数の入力と出力を詳細に説明するカスタム認可ロジックを有効にします。
3. IAM 認可: AWS署名バージョン 4 の署名プロセスを使用し、IAM ポリシーによるきめ細かなアクセスコントロールを可能にします。
4. OpenID Connect 認可: ユーザー認証のために OIDC 準拠のサービスと統合します。
5. Cognito ユーザープール: Cognito のユーザー管理機能を使用して、グループベースのアクセスコントロールを実装します。

認可タイプ

AWS AppSync GraphQL API を操作するためにアプリケーションを認可するには、5 つの方法があります。 AWS AppSync API または CLI コールで次のいずれかの認可タイプ値を指定することで、使用する認可タイプを指定します。

- **`API_KEY`**

  API キーを使用する場合。
- **`AWS_LAMBDA`**

  AWS Lambda 関数を使用する場合。
- **`AWS_IAM`**

  AWS Identity and Access Management （ IAM) アクセス許可を使用する場合。
- **`OPENID_CONNECT`**

  OpenID Connect プロバイダーを使用する場合。
- **`AMAZON_COGNITO_USER_POOLS`**

  Amazon Cognito ユーザープールを使用する場合。

これらの基本的な承認タイプは、ほとんどの開発者に有効です。より高度なユースケースでは、コンソール、CLI、 AWS CloudFormationを使用してさらに承認モードを追加できます。追加の認可モードの場合、 AWS AppSync は上記の値 (、、、API_KEYAWS_LAMBDAAWS_IAMOPENID_CONNECT、) を取得する認可タイプを提供しますAMAZON_COGNITO_USER_POOLS。

メインまたはデフォルトの承認タイプとしてAPI_KEY、AWS_LAMBDA、または AWS_IAM を指定した場合、それを追加の承認モードの 1 つとして再び指定することはできません。同様に、追加の承認モード間で API_KEY、AWS_LAMBDA、または AWS_IAM を重複して使用することはできません。複数の Amazon Cognito ユーザープールと OpenID Connect プロバイダーを使用できます。ただし、デフォルトの承認モードと追加の承認モード間で Amazon Cognito ユーザープールまたは OpenID Connect プロバイダーを重複して使用することはできません。Amazon Cognito ユーザープールまたは OpenID Connect プロバイダーに異なる複数のクライアントを設定する場合、対応する正規表現を使用できます。

API 設定に変更を保存すると、 は変更の伝達 AWS AppSync を開始します。設定変更が反映されるまで、 は AWS AppSync 引き続き以前の設定のコンテンツを提供します。設定変更が伝播されると、 AWS AppSync はすぐに新しい設定に基づいてコンテンツの提供を開始します。 AWS AppSync が API の変更を伝達している間、API が以前の設定または新しい設定に基づいてコンテンツを提供しているかどうかを判断できません。

API_KEY 認可

認証されていない API は、認証された API よりも厳格なスロットリングが必要になります。認証されていない GraphQL エンドポイントに対してスロットリングを制御する方法の 1 つは API キーを使用します。API キーは、認証されていない GraphQL エンドポイントを作成するときに AWS AppSync サービスによって生成されるアプリケーション内のハードコードされた値です。コンソール、CLI、または AWS AppSync API リファレンスから API キーを更新できます。

Console

1. にサインイン AWS Management Console し、AppSync コンソールを開きます。
   a. API ダッシュボードで、GraphQL API を選択します。
   b. サイドバーで [設定] を選択します。
2. デフォルト認可モードで、API キー を選択します。
3. [API キー] テーブルで、[API キーの追加] を選択します。

   新しい API キーがテーブルで生成されます。

- 古い API キーを削除するには、テーブル内で API キーを選択し、[削除] を選択します。

4. ページの最下部で [保存] をクリックします。

CLI

1. まだ設定していない場合は、 CLI AWS へのアクセスを設定します。詳細については、「設定の基本」を参照してください。
2. update-graphql-api コマンドを実行して GraphQL API オブジェクトを作成します。

   この特定のコマンドには次の 2 つのパラメータを入力する必要があります。

   1. GraphQL API の api-id。
   2. API の新しい name。同じ name を使用できます。
   3. API_KEY となる authentication-type。

   > **Note**
   >
   > Region など、設定する必要のあるパラメータは他にもありますが、通常はデフォルトで CLI 設定値になります。

   コマンドの例は、次のようになります。

```
aws appsync update-graphql-api --api-id abcdefghijklmnopqrstuvwxyz --name
 TestAPI --authentication-type API_KEY
```

   出力は CLI に返されます。以下に JSON の例を示します。

```
{
    "graphqlApi": {
        "xrayEnabled": false,
        "name": "TestAPI",
        "authenticationType": "API_KEY",
        "tags": {},
        "apiId": "abcdefghijklmnopqrstuvwxyz",
        "uris": {
```

```
            "GRAPHQL": "https://s8i3kk3ufhe9034ujnv73r513e.appsync-api.us-
west-2.amazonaws.com/graphql",
            "REALTIME": "wss://s8i3kk3ufhe9034ujnv73r513e.appsync-realtime-
api.us-west-2.amazonaws.com/graphql"
        },
        "arn": "arn:aws:appsync:us-west-2:348581070237:apis/
abcdefghijklmnopqrstuvwxyz"
    }
}
```

API キーは、最大 365 日間有効に設定可能で、該当日からさらに最大 365 日、既存の有効期限を延長できます。API キーは、パブリック API の公開が安全であるユースケース、または開発目的での使用が推奨されます。

クライアントでは、API キーをヘッダー `x-api-key` で指定します。

たとえば、`API_KEY` が `'ABC123'` である場合、次のように `curl` 経由で GraphQL クエリを送信できます。

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "x-api-key:ABC123" -d
 '{ "query": "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

AWS_LAMBDA 認可

AWS Lambda 機能を使用して、独自の API 認可ロジックを実装できます。Lambda 関数をプライマリオーソライザーまたはセカンダリオーソライザーに使用できますが、API ごとに Lambda 認可関数は 1 つしかありません。Lambda 関数を認可に使用する場合、次のことが適用されます。

- API で `AWS_LAMBDA` および `AWS_IAM` 認可モードが有効になっている場合、SigV4 署名を `AWS_LAMBDA` 認可トークンとして使用することはできません。
- API で `AWS_LAMBDA` および `OPENID_CONNECT` 認可モードまたは `AMAZON_COGNITO_USER_POOLS` 認可モードが有効になっている場合、OIDC 署名を `AWS_LAMBDA` 認可トークンとして使用することはできません。OIDC トークンはベアラースキームにすることができます。
- Lambda 関数は、リゾルバーに対して 5MB を超えるコンテキストデータを返してはいけません。

例えば、認可トークン が `'ABC123'` である場合、次のように curl 経由で GraphQL クエリを送信できます。

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "Authorization:ABC123" -d
 '{ "query":
        "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

Lambda 関数は、各クエリまたはミューテーションの前に呼び出されます。戻り値は API ID と認証トークンに基づいてキャッシュできます。Lambda オーソライザーのレスポンスが 1,048,576 バイト未満の場合、 AWS AppSync は後続のリクエストのレスポンスをキャッシュします。Lambda オーソライザーのレスポンスが 1,048,576 バイト以上の場合、 AWS AppSync はレスポンスをキャッシュせず、受信リクエストごとに Lambda オーソライザーを呼び出します。パフォーマンスを最適化し、Lambda 呼び出しコストを最小限に抑えるには、Lambda オーソライザーのレスポンスを 1,048,576 バイトに制限することをお勧めします。デフォルトでは、キャッシュは有効になっていませんが、これは API レベルで、または関数の戻り値の `ttlOverride` の値を設定することで有効にすることができます。

必要に応じて、関数が呼び出される前に認可トークンを検証する正規表現を指定できます。これらの正規表現は、関数が呼び出される前に、認可トークンが正しい形式であることを検証するために使用されます。この正規表現と一致しないトークンを使用したリクエストは、自動的に拒否されます。

認可に使用される Lambda 関数には、 `appsync.amazonaws.com` がそれらを AWS AppSync 呼び出すことができるように、 のプリンシパルポリシーを適用する必要があります。このアクションは AWS AppSync コンソールで自動的に行われます。 AWS AppSync コンソールはポリシーを削除しません。Lambda 関数にポリシーをアタッチする方法の詳細については、「 AWS Lambda デベロッパーガイド」の「リソースベースのポリシー」を参照してください。

指定した Lambda 関数は次の形状のイベントを受け取ります。

```
{
    "authorizationToken": "ExampleAUTHtoken123123123",
    "requestContext": {
        "apiId": "aaaaaa123123123example123",
        "accountId": "111122223333",
        "requestId": "f4081827-1111-4444-5555-5cf4695f339f",
        "queryString": "mutation CreateEvent {...}\n\nquery MyQuery {...}\n",
        "operationName": "MyQuery",
        "variables": {}
    }
    "requestHeaders": {
        application request headers
    }
```

```
}
```

event オブジェクトには、アプリケーションクライアントから へのリクエストで送信されたヘッダーが含まれています AWS AppSync。

認可関数は少なくとも を返す必要があります。これはisAuthorized、リクエストが認可されているかどうかを示すブール値です。Lambda 認可関数から返された次のキー AWS AppSync を認識します。

> **Note**
>
> WebSocket 接続オペレーションrequestContextの operationNameの の値は、 によってDeepDish:Connect AWS AppSync 「」に設定されます。

関数のリスト

isAuthorized (boolean、必須)

GraphQL API への呼び出しをおこなうために authorizationToken の値が認証されるかどうかを示すブール値。

この値が true の場合、GraphQL API の実行が継続されます。この値が false の場合、UnauthorizedException が生成されます。

deniedFields (文字列のリスト、オプション)

リゾルバーから値が返された場合でも、そのリストは強制的に null に変更されます。

各項目は、arn:aws:appsync:*us-east-1*:*111122223333*:apis/*GraphQLApiId*/types/*TypeName*/fields/*FieldName* の形式、または *TypeName*.*FieldName* の短い形式の完全修飾フィールド ARN です。2 つの API が Lambda 関数のオーソライザーを共有し、2 つの API の間に共通の型とフィールドの間にあいまいさがある可能性がある場合は、完全な ARN 形式を使用する必要があります。

resolverContext (JSON オブジェクト、オプション)

リゾルバーテンプレート内の $ctx.identity.resolverContext として可視化されるJSON オブジェクト。たとえば、リゾルバーによって次の構造体が返されたとします。

```
{
  "isAuthorized":true
```

```
  "resolverContext": {
    "banana":"very yellow",
    "apple":"very green"
  }
}
```

リゾルバーテンプレート内の値 `ctx.identity.resolverContext.apple` は「very green」になります。`resolverContext` オブジェクト はキーと値のペアのみをサポートします。ネストされたキーはサポートされません。

> ⚠ Warning
>
> この JSON オブジェクトの合計サイズは 5 MB を超えないようにしてください。

`ttlOverride` (integer、オプション)

応答をキャッシュする秒数。値が返されない場合は、API からの値が使用されます。これが 0 の場合、応答はキャッシュされません。

Lambda オーソライザーのタイムアウトは 10 秒です。API のパフォーマンスをスケールするために、できるだけ短い時間で実行する関数を設計することをお勧めします。

AWS AppSync APIsは、単一の認証 Lambda 関数を共有できます。クロスアカウントオーソライザーの使用は許可されていません。

複数の API 間で認可関数を共有する場合は、短い形式のフィールド名 (*typename*.*fieldname*) が誤ってフィールドを隠すことがあります。`deniedFields` のフィールドを明確化するには、明確なフィールド ARN を `arn:aws:appsync:`*region*`:`*accountId*`:apis/`*GraphQLApiId*`/types/`*typeName*`/fields/`*fieldName* の形式で指定できます。

AWS AppSyncで Lambda 関数をデフォルトの認可モードとして追加するには、

Console

1. AWS AppSync コンソールにログインし、更新する API に移動します。
2. API の設定ページに移動します。

   API レベルの認可をAWS Lambdaに変更します。
3. API コールを許可する AWS リージョン と Lambda ARN を選択します。

> **Note**
>
> 適切なプリンシパルポリシーが自動的に追加され、 AWS AppSync Lambda 関数を呼び出します。

4. 必要に応じて、レスポンス TTL とトークン検証の正規表現を設定します。

AWS CLI

1. 使用中の Lambda 関数に次のポリシーをアタッチします。

```
aws lambda add-permission --function-name "my-function" --statement-id "appsync"
 --principal appsync.amazonaws.com --action lambda:InvokeFunction --output text
```

> **Important**
>
> 関数のポリシーを単一の GraphQL API にロックする場合は、次のコマンドを実行できます。
>
> ```
> aws lambda add-permission --function-name “my-function” --
> statement-id “appsync” --principal appsync.amazonaws.com --action
>  lambda:InvokeFunction --source-arn “<my AppSync API ARN>” --output text
> ```

2. 指定された Lambda 関数 ARN をオーソライザーとして使用するように AWS AppSync API を更新します。

```
aws appsync update-graphql-api --api-id example2f0ur2oid7acexample --
name exampleAPI --authentication-type AWS_LAMBDA --lambda-authorizer-config
 authorizerUri="arn:aws:lambda:us-east-2:111122223333:function:my-function"
```

> **Note**
>
> トークンの正規表現など、他の設定オプションを含めることもできます。

次の例では、Lambda 関数が AWS AppSync 認可メカニズムとして使用されたとき、その Lambda 関数が持つさまざまな認証状態および認証失敗状態を示しています。

```
def handler(event, context):
  # This is the authorization token passed by the client
  token = event.get('authorizationToken')
  # If a lambda authorizer throws an exception, it will be treated as unauthorized.
  if 'Fail' in token:
    raise Exception('Purposefully thrown exception in Lambda Authorizer.')

  if 'Authorized' in token and 'ReturnContext' in token:
    return {
      'isAuthorized': True,
      'resolverContext': {
        'key': 'value'
      }
    }

  # Authorized with no f
  if 'Authorized' in token:
    return {
      'isAuthorized': True
    }
  # Partial authorization
  if 'Partial' in token:
    return {
      'isAuthorized': True,
      'deniedFields':['user.favoriteColor']
    }
  if 'NeverCache' in token:
    return {
      'isAuthorized': True,
      'ttlOverride': 0
    }
  if 'Unauthorized' in token:
    return {
      'isAuthorized': False
    }
  # if nothing is returned, then the authorization fails.
  return {}
```

SigV4 と OIDC トークンの認可制限を回避する

以下の方法を使用すると、特定の認可モードが有効になっている場合に SigV4 署名または OIDC トークンを Lambda 認可トークンとして使用できないという問題を回避できます。

AWS AppSyncの API に対して `AWS_IAM` および `AWS_LAMBDA` 認可モードが有効になっているとき、SigV4 署名を Lambda 認可トークンとして使用する場合は、次の操作を行います。

- 新しい Lambda 認可トークンを作成するには、SigV4 署名にランダムなサフィックスおよび/またはプレフィックスを追加します。
- 元の SigV4 署名を取得するには、Lambda 認可トークンからランダムなプレフィックスおよび/またはサフィックスを削除して、Lambda 関数を更新します。次に、元の SigV4 署名を認証に使用します。

の AWS AppSync API で認可モードまたは `AMAZON_COGNITO_USER_POOLS`および `OPENID_CONNECT`認可モードが有効になっているときに、OIDC トークンを Lambda `AWS_LAMBDA`認可トークンとして使用する場合は、以下を実行します。

- 新しい Lambda 認可トークンを作成するには、OIDC トークンにランダムなサフィックスおよび/またはプレフィックスを追加します。Lambda 認可トークンにはベアラースキームプレフィックスを含めないでください。
- 元の OIDC トークンを取得するには、Lambda 認可トークンからランダムなプレフィックスおよび/またはサフィックスを削除して、Lambda 関数を更新します。次に、元の OIDC トークンを認証に使用します。

AWS_IAM 認可

この承認タイプでは、GraphQL API に対して AWS 署名バージョン 4 署名プロセスを使用する必要があります。Identity and Access Management (IAM) アクセスポリシーをこの承認タイプに関連付けることができます。アクセスキー (アクセスキー ID とシークレットアクセスキーで構成) または Amazon Cognito フェデレーティッドアイデンティティによって提供される有効期限の短い、一時的な認証情報を使用して、アプリケーションでこの関連付けを活用します。

すべてのデータオペレーションを実行できるアクセス権限を持つロールが必要な場合。

JSON

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "appsync:GraphQL"
         ],
         "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/*"
         ]
      }
   ]
}
```

AppSync コンソールのメイン API リストページから、使用する API の名前の直下で、YourGraphQLApiId を確認できます。CLI `aws appsync list-graphql-apis` を使用して取得することもできます。

特定の GraphQL オペレーションのみにアクセスを制限するには、ルートの Query、Mutation、Subscription の各フィールドに対してこれを実行します。

JSON

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "appsync:GraphQL"
         ],
         "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/
Query/fields/<Field-1>",
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/
Query/fields/<Field-2>",
```

```
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/
Mutation/fields/<Field-1>",
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/
Subscription/fields/<Field-1>"
         ]
     }
   ]
}
```

たとえば、以下のスキーマがあり、すべての投稿を取得するアクセスを制限する場合。

```
schema {
   query: Query
   mutation: Mutation
}

type Query {
   posts:[Post!]!
}

type Mutation {
   addPost(id:ID!, title:String!):Post!
}
```

ロール (Amazon Cognito ID プールなどにアタッチできる) に対応する IAM ポリシーは次のようになります。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
            "appsync:GraphQL"
            ],
            "Resource": [
                "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/
types/Query/fields/posts"
            ]
```

```
        }
    ]
}
```

OPENID_CONNECT 認可

この認可タイプは、OIDC 準拠サービスによって提供される OpenID Connect (OIDC) トークンを適用します。アプリケーションは、アクセス制御に対して、使用する OIDC プロバイダーによって定義されたユーザーと権限を活用できます。

発行者 URL は、 AWS AppSync に提供する唯一の必須設定値です (例: `https://auth.example.com`)。この URL は HTTPS 経由でアドレス可能である必要があります。 AWS AppSync は発行者 URL `/.well-known/openid-configuration` に を追加し、OpenID Connect Discovery 仕様`https://auth.example.com/.well-known/openid-configuration`に従ってで OpenID 設定を見つけます。 OpenID この URL で RFC5785 準拠の JSON ドキュメントを取得するものとします。この JSON ドキュメントには、署名`jwks_uri`キーを含む JSON Web Key Set (JWKS) ドキュメントを指すキーが含まれている必要があります。 AWS AppSync では、JWKS に `kty`および の JSON フィールドが含まれている必要があります`kid`。

AWS AppSync は、さまざまな署名アルゴリズムをサポートしています。

署名アルゴリズム
RS256
RS384
RS512
PS256
PS384
PS512
HS256
HS384

署名アルゴリズム
HS512
ES256
ES384
ES512

RSA アルゴリズムを使用することをお勧めします。プロバイダーによって発行されたトークンに、トークンが発行された時刻 (iat) が含まれている必要があり、認証された時刻 (auth_time) が含まれる場合があります。発行時刻の TTL 値 (iatTTL) および認証時刻の TTL 値 (authTTL) を、追加の検証用に OpenID Connect 設定に入力できます。使用するプロバイダーが複数のアプリケーションを認可している場合は、クライアント ID で認可するために使用される正規表現を (clientId) も入力できます。clientId が OpenID Connect 設定に存在する場合、 AWS AppSync は、 がトークンの audまたは クレームのいずれかと一致するclientIdように要求することでazp、クレームを検証します。

複数のクライアント ID を検証するには、正規表現で「or」であるパイプライン演算子 ("|") を使用します。例えば、OIDC アプリケーションに 4 つのクライアントがあり、そのクライアント ID が 0A1S2D、1F4G9H、1J6L4B、6GS5MG などである場合、最初の 3 つのクライアント ID のみを検証するには、クライアント ID フィールドに「1F4G9H|1J6L4B|6GS5MG」と入力します。

API が複数の認可タイプで設定されている場合、 AWS AppSync はリクエストヘッダーから JWT トークンに存在する発行者 (iss claim) を API 設定で指定された発行者 URL と比較することで検証します。ただし、API が OPENID_CONNECT 認可のみを使用して設定されている場合、 AWS AppSync はこの発行者 URL 検証ステップをスキップします。

AMAZON_COGNITO_USER_POOLS 認可

この認可タイプでは、Amazon Cognito ユーザープールによって提供される OIDC トークンが使用されます。アプリケーションは、別のアカウントのユーザープールとユーザープールの両方のユーザーとグループを活用し AWS 、これらを GraphQL フィールドに関連付けてアクセスを制御できます。

Amazon Cognito ユーザープールを使用する場合、ユーザーが属するグループを作成できます。この情報は、アプリケーションが GraphQL オペレーションを送信するときに認可ヘッダーで AWS AppSync に送信する JWT トークンにエンコードされます。スキーマで GraphQL ディレクティブを

使用して、フィールドでどのグループがどのリゾルバーを起動できるのかを制御します。したがってカスタマーのアクセスを細かく制御できます。

たとえば、以下の GraphQL スキーマがあるとします。

```
schema {
   query: Query
   mutation: Mutation
}

type Query {
   posts:[Post!]!
}

type Mutation {
   addPost(id:ID!, title:String!):Post!
}
...
```

Amazon Cognito ユーザープールに 2 つのグループ (bloggers と readers) があり、readers が新しいエントリを追加できないように制限する場合、スキーマは次のようになります。

```
schema {
   query: Query
   mutation: Mutation
}
```

```
type Query {
   posts:[Post!]!
   @aws_auth(cognito_groups: ["Bloggers", "Readers"])
}

type Mutation {
   addPost(id:ID!, title:String!):Post!
   @aws_auth(cognito_groups: ["Bloggers"])
}
...
```

@aws_auth ディレクティブは、個別のアクセス許可/拒否戦略をデフォルトにする場合に省略できることに注意してください。コンソールまたは以下の CLI コマンド経由で GraphQL API を作成するときに、ユーザープール設定で許可/拒否戦略を指定できます。

```
$ aws appsync --region us-west-2 create-graphql-api --authentication-
type AMAZON_COGNITO_USER_POOLS  --name userpoolstest --user-pool-config
 '{ "userPoolId":"test", "defaultEffect":"ALLOW", "awsRegion":"us-west-2"}'
```

追加の承認モードの使用

追加の認可モードを追加すると、 AWS AppSync GraphQL API レベル (GraphqlApiオブジェクトで直接設定できる authenticationTypeフィールド) で認可設定を直接設定でき、スキーマのデフォルトとして機能します。つまり、特定のディレクティブのないタイプは API レベルの承認設定を渡す必要があります。

スキーマレベルでは、スキーマでディレクティブを使用して追加の承認モードを指定できます。スキーマの個々のフィールドに承認モードを指定できます。たとえば、API_KEY 承認の場合、スキーマオブジェクトタイプの定義/フィールドで @aws_api_key を使用します。以下のディレクティブは、スキーマフィールドおよびオブジェクトタイプ定義でサポートされています。

- @aws_api_key - フィールドが API_KEY で承認されることを指定します。
- @aws_iam - フィールドが AWS_IAM で承認されることを指定します。
- @aws_oidc - フィールドが OPENID_CONNECT で承認されることを指定します。
- @aws_cognito_user_pools - フィールドが AMAZON_COGNITO_USER_POOLS で承認されることを指定します。
- @aws_lambda - フィールドが AWS_LAMBDA で承認されることを指定します。

@aws_auth ディレクティブを追加の承認モードと共に使用することはできません。@aws_auth は、追加の承認モードのない AMAZON_COGNITO_USER_POOLS 承認のコンテキストでのみ機能します。ただし、同じ引数で @aws_auth ディレクティブの代わりに @aws_cognito_user_pools ディレクティブを使用できます。2 つの主な違いは、フィールドとオブジェクトタイプの定義で @aws_cognito_user_pools を指定できることです。

追加の承認モードがどのように機能し、スキーマでどのように指定できるかを理解するために、以下のスキーマを見てみましょう。

```
schema {
   query: Query
   mutation: Mutation
}
```

```
type Query {
   getPost(id: ID): Post
   getAllPosts(): [Post]
   @aws_api_key
}

type Mutation {
   addPost(
      id: ID!
      author: String!
      title: String!
      content: String!
      url: String!
   ): Post!
}

type Post @aws_api_key @aws_iam {
   id: ID!
   author: String
   title: String
   content: String
   url: String
   ups: Int!
   downs: Int!
   version: Int!
}
...
```

このスキーマでは、 AWS_IAMが AWS AppSync GraphQL API のデフォルトの認可タイプであると仮定します。つまり、ディレクティブのないフィールドは AWS_IAM を使用して保護されます。たとえば、Query タイプの getPost フィールドの場合です。スキーマディレクティブにより、複数の承認モードを使用できます。たとえば、 AWS AppSync GraphQL API で を追加の認可モードとしてAPI_KEY設定し、 @aws_api_keyディレクティブ (getAllPostsこの例では など) を使用してフィールドをマークできます。ディレクティブはフィールドレベルで機能するため、Post タイプへの API_KEY アクセスも許可する必要があります。そのためには、Post タイプの各フィールドをディレクティブでマークするか、Post タイプを @aws_api_key ディレクティブでマークします。

Post タイプのフィールドへのアクセスをさらに制限するには、以下に示すように、Post タイプの個々のフィールドに対してディレクティブを使用できます。

たとえば、restrictedContent フィールドを Post タイプに追加し、@aws_iam ディレクティブを使用してそのフィールドへのアクセスを制限できます。ただし restrictedContent

に、AWS_IAM 認証済みリクエストからはアクセスできますが、API_KEY リクエストからはアクセスできません。

```
type Post @aws_api_key @aws_iam{
   id: ID!
   author: String
   title: String
   content: String
   url: String
   ups: Int!
   downs: Int!
   version: Int!
   restrictedContent: String!
   @aws_iam
}
...
```

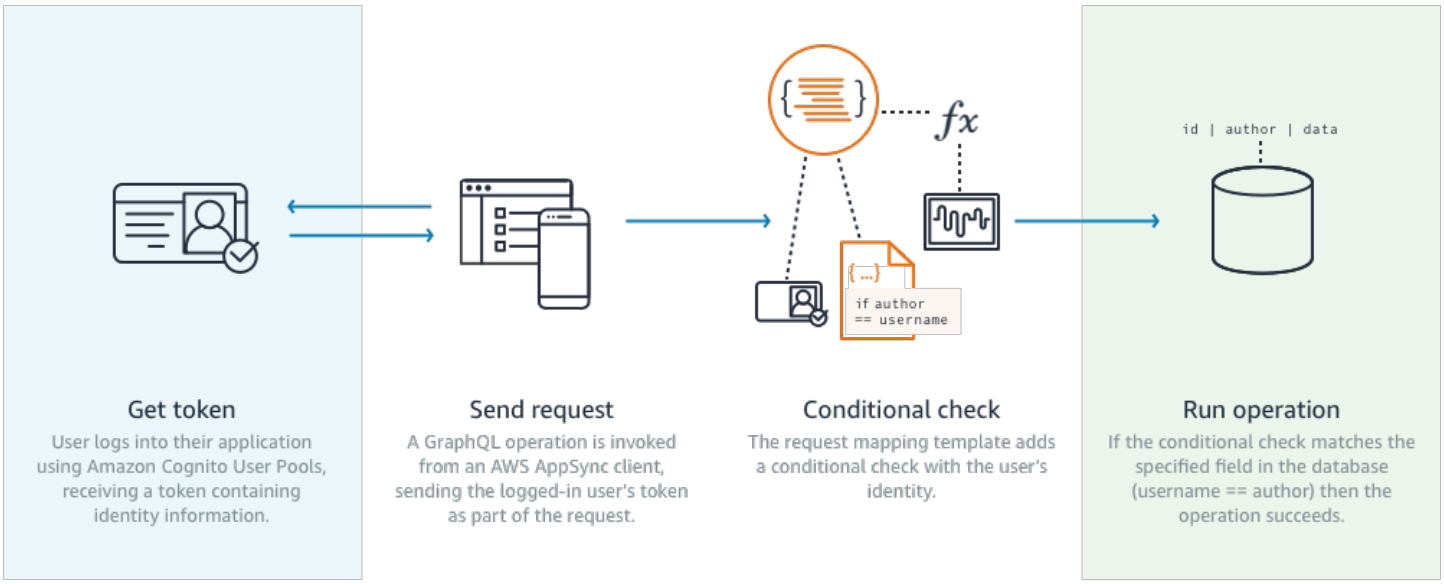
きめ細かなアクセスコントロール

前述の情報は、特定の GraphQL フィールドへのアクセスを制限または許可する方法を示しています。特定の条件に基づいて (たとえば、呼び出し元のユーザーがだれであるかや、そのユーザーがデータを所有しているかどうかに基づいて) データに対するアクセスコントロールを設定する場合は、リゾルバーでマッピングテンプレートを使用できます。より複雑なビジネスロジックも実行できます。「フィルタ処理情報」で説明します。

このセクションでは、DynamoDB リゾルバーマッピングテンプレート使用してデータのアクセスコントロールを設定する方法を示します。

さらに進む前に、 AWS AppSync のマッピングテンプレートがよくわからない場合は、「リゾルバーのマッピングテンプレートリファレンス」と「DynamoDB のリゾルバーのマッピングテンプレートリファレンス」を参照してください。

DynamoDB を使用した次の例では、前述のブログ投稿スキーマを使用し、投稿を作成したユーザーのみが編集を許可されているものとします。評価プロセスは、Amazon Cognito ユーザープールなどを使用して、ユーザーがアプリケーションで認証情報を取得し、GraphQL オペレーションの一部として、これらの認証情報を渡すというものです。その後、マッピングテンプレートが、条件ステートメントで、認証情報 (username など) からの値を置き換えます。値はデータベースの値と比較されます。



この機能を追加するには、次のように editPost GraphQL フィールドを追加します。

```
schema {
   query: Query
   mutation: Mutation
}

type Query {
   posts:[Post!]!
}

type Mutation {
   editPost(id:ID!, title:String, content:String):Post
   addPost(id:ID!, title:String!):Post!
}
...
```

editPost のリゾルバーマッピングテンプレート (このセクションの最後にある例) では、投稿を作成したユーザーのみが編集を許可されるように、データストアに対する論理チェックを実行する必要があります。これは編集オペレーションなので、DynamoDB の UpdateItem に対応します。ユーザー ID 検証に渡されるコンテキストを使用して、このアクションを実行する前に条件チェックを実行できます。これは次の値を持つ Identity オブジェクトに保存されます。

```
{
   "accountId" : "12321434323",
   "cognitoIdentityPoolId" : "",
```

```
    "cognitoIdentityId" : "",
    "sourceIP" : "",
    "caller" : "ThisistheprincipalARN",
    "username" : "username",
    "userArn" : "Sameasabove"
}
```

DynamoDB UpdateItem コールでこのオブジェクトを使用するには、比較用テーブルにユーザー ID 情報を保存する必要があります。まず、addPost ミューテーションは作成者を保存する必要があります。次に、更新する前に、editPost ミューテーションで条件チェックを実行する必要があります。

Author 列としてユーザー ID を保存する addPost のリゾルバーコードの例を次に示します。

```
import { util, Context } from '@aws-appsync/utils';
import { put } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const { id: postId, ...item } = ctx.args;
 return put({
  key: { postId },
  item: { ...item, Author: ctx.identity.username },
  condition: { postId: { attributeExists: false } },
 });
}

export const response = (ctx) => ctx.result;
```

Author 属性が、アプリケーションから得られた Identity オブジェクトから入力されていることに注意してください。

最後に、editPost のリゾルバーコードの例を次に示します。これは、投稿を作成したユーザーからリクエストが来た場合にのみ、ブログ投稿のコンテンツを更新します。

```
import { util, Context } from '@aws-appsync/utils';
import { put } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const { id, ...item } = ctx.args;
 return put({
  key: { id },
```

```
  item,
  condition: { author: { contains: ctx.identity.username } },
 });
}

export const response = (ctx) => ctx.result;
```

この例では、UpdateItem ではなく、すべての値を上書きする PutItem を使用していますが、condition ステートメントブロックには同じ概念が適用されます。

フィルタ処理情報

データソースからのレスポンスを制御できないときに、データソースへの正常な書き込みまたは読み取りに対して、不必要な情報をクライアントに送信したくない場合があります。このような場合は、レスポンスマッピングテンプレートを使用して情報をフィルタリングすることができます。

たとえば、ブログ投稿 DynamoDB テーブルに適切なインデックス (Author のインデックスなど) がない場合を考えます。以下のリゾルバーを使用できます。

```
import { util, Context } from '@aws-appsync/utils';
import { get } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 return get({ key: { ctx.args.id } });
}

export function response(ctx) {
 if (ctx.result.author === ctx.identity.username) {
  return ctx.result;
 }
 return null;
}
```

リクエストハンドラは、呼び出し元が投稿の作成者でなくても、項目をフェッチします。これによってすべてのデータが返されるのを防ぐために、レスポンスハンドラは呼び出し元が項目の作成者と一致することを確認します。発信者がこのチェックに一致しない場合に、null レスポンスのみが返されます。

データソースへのアクセス

AWS AppSync は、Identity and Access Management (IAM) ロールとアクセスポリシーを使用してデータソースと通信します。既存のロールを使用している場合、 AWS AppSync がロールを引き受けるには、信頼ポリシーを追加する必要があります。信頼関係は以下のようになります。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "appsync.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

必要な最小限のリソースについて作業を行うアクセス許可のみを持つように、ロールのアクセスポリシーを制限することが重要です。AppSync コンソールを使用してデータソースを作成し、ロールを作成したとき、これは自動的に行われます。ただし、IAM コンソールから組み込みサンプルテンプレートを使用して AWS AppSync コンソールの外部でロールを作成する場合、アクセス許可はリソースに対して自動的にスコープダウンされないため、アプリケーションを本番環境に移行する前にこのアクションを実行する必要があります。

リクエストとレスポンスを保護するためのアクセスコントロールのユースケース

「セキュリティ」セクションでは、API を保護するためのさまざまな認可モードについて説明し、きめ細かな認可メカニズムの概念と流れについて紹介します。 AWS AppSync を使用すると、GraphQL Resolver Mapping テンプレートを使用してデータに対してロジックフルオペレーションを実行できるため、ユーザー ID、条件、データインジェクションの組み合わせを使用して、読み取りまたは書き込み時のデータを非常に柔軟に保護できます。

AWS AppSync Resolver の編集に慣れていない場合は、プログラミングガイドを参照してください。

概要

システム内のデータへのアクセスの付与は、従来、行 (リソース) と列 (ユーザーまたはロール) の交点が付与されるアクセス許可である アクセスコントロールマトリックス を通じて行われています。

AWS AppSync は独自のアカウントのリソースを使用し、アイデンティティ (ユーザー/ロール) 情報をコンテキストオブジェクトとして GraphQL リクエストとレスポンスにスレッドします。これはリゾルバーで使用できます。つまり、リゾルバーのロジックに基づいて、書き込みまたは読み取りのオペレーションに対して適切にアクセス権限を付与できます。このロジックがリソースレベルで、たとえば、特定の名前付きユーザーまたはグループのみが特定のデータベース行に読み書きできる場合、その「認可メタデータ」を保存する必要があります。 AWS AppSync はデータを保存しないため、アクセス許可を計算できるように、この認可メタデータをリソースに保存する必要があります。認可メタデータは通常、DynamoDB テーブル内の属性 (列) であり、所有者、ユーザーまたはグループのリストなどです。たとえば、Readers と Writers 属性があります。

ハイレベルでは、データソースから個々の項目を読み取っている場合、リゾルバーがデータソースから読み取った後に、レスポンステンプレートで条件ステートメント `#if () ... #end` を実行します。そのチェックでは通常、読み取りオペレーションから返された認可メタデータに対するメンバーシップ確認のために、`$context.identity` の user または group の値が使用されます。複数のレコードがある (テーブルの `Scan` や `Query` で返されるリストなど) 場合は、同様の user または group の値を使用して、データソースに対するオペレーションの一部として条件チェックを送信します。

同様に、データを書き込む場合は、アクション (ミューテーションを作成するユーザーやグループにアクセス権限があるかどうかを確認する `PutItem` や `UpdateItem` など) に対して条件ステートメントを適用します。この条件チェックでは、`$context.identity` 内の値を使用して何度も行われ、リソースの認可メタデータと比較されます。リクエストテンプレートとレスポンステンプレートの両方で、クライアントからのカスタムヘッダーを使用して検証チェックを実行することもできます。

データの読み込み

前述のように、チェックを実行するための認可メタデータは、リソースに保存されているかまたは GraphQL リクエスト (ID、ヘッダーなど) で渡される必要があります。その例を示すために、次の DynamoDB テーブルがあるとします。

ID	Data	PeopleCanAccess	GroupsCanAccess	Owner
123	{my: data,...}	[Mary, Joe]	[Admins, Editors]	Nadia

プライマリキーは id であり、アクセスするデータは Data です。その他の列は、認可のために実行できるチェックの例です。「DynamoDB のリゾルバーのマッピングテンプレートリファレンス」に概説されている通り、Owner は String になり、PeopleCanAccess と GroupsCanAccess は String Sets になります。

「リゾルバーのマッピングテンプレートの概要」の図に示しているように、レスポンステンプレートには、コンテキストオブジェクトだけでなくデータソースからの結果も含まれています。個々の項目に対する GraphQL クエリでは、レスポンステンプレートを使用して、そのユーザーが結果を確認することを許可されているかどうかをチェックし、そうでない場合は認可エラーメッセージを返すことができます。これは、「認可フィルタ」と呼ばれることもあります。Scan または Query を使用してリストを返す GraphQL クエリでは、リクエストテンプレートでチェックを実行し、認可条件が満たされている場合にのみデータを返すのが、より効率的です。次のように実装します。

1. GetItem - 個々のレコードに対する認可チェック。#if( ) ... #end ステートメントを使用して行われます。
2. Scan/Query オペレーション - 認可チェックは "filter":{"expression":...} ステートメントです。よく使用されるチェックは、等価チェック (attribute = :input)、または値がリストあるかどうかのチェック (contains(attribute, :input)) です。

上記の 2 で両方のステートメントにある attribute は、テーブル内のレコードの列名 (上記の例では Owner など) を表しています。そのエイリアスとして # 記号と "expressionNames":{...} を使用できますが、必須ではありません。:input は、データベース属性と比較する値への参照であり、"expressionValues":{...} で定義します。その例を以下に示します。

ユースケース: 所有者が読み取り可能

上記のテーブルを使用して、個々の読み取りオペレーション (Owner == Nadia) で GetItem であるときにのみデータを返す場合、テンプレートは次のようになります。

```
#if($context.result["Owner"] == $context.identity.username)
    $utils.toJson($context.result)
#else
```

```
    $utils.unauthorized()
#end
```

ここで、以降のセクションで再利用する点について説明しておきます。まず、チェックで使用される `$context.identity.username` は、Amazon Cognito ユーザープールが使用されている場合は分かりやすいユーザーサインアップ名であり、IAM (Amazon Cognito フェデレーテッドアイデンティティも含む) が使用されている場合はユーザー ID です。所有者に対して保存されるその他の値として、複数の口ケーションからフェデレーテッドログインする場合に便利な一意の「Amazon Cognito ID」値などがあり、「リゾルバーのマッピングテンプレートのコンテキストリファレンス」で、利用可能なオプションを確認しておく必要があります。

次に、`$util.unauthorized()` に対応する条件付き else チェックは、完全に省略可能ですが、GraphQL API を設計する際のベストプラクティスとして、指定することお勧めします。

ユースケース: 特定のアクセス権のハードコード

```
// This checks if the user is part of the Admin group and makes the call
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #if($group == "Admin")
        #set($inCognitoGroup = true)
    #end
#end
#if($inCognitoGroup)
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "attributeValues" : {
        "owner" : $util.dynamodb.toDynamoDBJson($context.identity.username)
        #foreach( $entry in $context.arguments.entrySet() )
            ,"${entry.key}" : $util.dynamodb.toDynamoDBJson($entry.value)
        #end
    }
}
#else
    $utils.unauthorized()
#end
```

ユースケース: 結果リストのフィルタリング

前の例では単一の項目が返されるため、$context.result に対してチェックを直接実行することもできましたが、スキャンなどの一部のオペレーションでは $context.result.items で複数の項目が返されるため、認可フィルタを実行して、そのユーザーが確認を許可されている結果のみを返す必要があります。たとえば、今度はレコードに設定されている Amazon Cognito IdentityID が Owner フィールドにあるとすると、次のレスポンスマッピングテンプレートを使用して、そのユーザーが所有しているレコードのみを示すようにフィルタリングできます。

```
#set($myResults = [])
#foreach($item in $context.result.items)
    ##For userpools use $context.identity.username instead
    #if($item.Owner == $context.identity.cognitoIdentityId)
        #set($added = $myResults.add($item))
    #end
#end
$utils.toJson($myResults)
```

ユースケース: 複数のユーザーが読み取り可能

もう 1 つのよくある認可オプションは、ユーザーのグループがデータを読み取ることができるように許可することです。次の例の "filter":{"expression":...} では、GraphQL クエリを実行しているユーザーが PeopleCanAccess のセットにリストされている場合にのみ、テーブルスキャンからの値が返されます。

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "limit": #if(${context.arguments.count}) $util.toJson($context.arguments.count)
 #else 20 #end,
    "nextToken": #if(${context.arguments.nextToken})
 $util.toJson($context.arguments.nextToken) #else null #end,
    "filter":{
        "expression": "contains(#peopleCanAccess, :value)",
        "expressionNames": {
                "#peopleCanAccess": "peopleCanAccess"
        },
        "expressionValues": {
                ":value": $util.dynamodb.toDynamoDBJson($context.identity.username)
        }
    }
```

```
}
```

ユースケース: グループが読み取り可能

直前のユースケースと同様に、1 つまたは複数のグループに属するユーザーのみが、データベース内の特定の項目を読み取る権限を持っているとします。"expression": "contains()" オペレーションを使用するのは同じですが、設定されているメンバーシップに属している必要があるすべてのグループの論理 OR です。この例では、ユーザーが属している各グループに対して次の $expression ステートメントを作成し、それをフィルタに渡します。

```
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
    #set( $val = {})
    #set( $test = $val.put("S", $group))
    #set( $values = $expressionValues.put(":var$foreach.count", $val))
    #if ( $foreach.hasNext )
    #set( $expression = "${expression} OR" )
    #end
#end
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "limit": #if(${context.arguments.count}) $util.toJson($context.arguments.count)
 #else 20 #end,
    "nextToken": #if(${context.arguments.nextToken})
 $util.toJson($context.arguments.nextToken) #else null #end,
    "filter":{
        "expression": "$expression",
        "expressionValues": $utils.toJson($expressionValues)
    }
}
```

データの書き込み

ミューテーションでのデータの書き込みは、常にリクエストマッピングテンプレートで制御されます。DynamoDB のデータソースの場合、key は、そのテーブル内の認可メタデータに対して検証を実行する適切な "condition":{"expression"...}" が使用されます。「セキュリティ」には、

テーブル内の Author フィールドのチェックに役立つ例があります。このセクションでは、その他のユースケースを示します。

ユースケース: 複数の所有者

以前の例のテーブル図を使用した、PeopleCanAccess リストがあるとします。

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "update" : {
        "expression" : "SET meta = :meta",
        "expressionValues": {
            ":meta" : $util.dynamodb.toDynamoDBJson($ctx.args.meta)
        }
    },
    "condition" : {
        "expression"       : "contains(Owner,:expectedOwner)",
        "expressionValues" : {
            ":expectedOwner" :
 $util.dynamodb.toDynamoDBJson($context.identity.username)
        }
    }
}
```

ユースケース: グループが新規レコードを作成可能

```
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
    #set( $val = {})
    #set( $test = $val.put("S", $group))
    #set( $values = $expressionValues.put(":var$foreach.count", $val))
    #if ( $foreach.hasNext )
    #set( $expression = "${expression} OR" )
    #end
#end
{
```

```
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        ## If your table's hash key is not named 'id', update it here. **
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
        ## If your table has a sort key, add it as an item here. **
    },
    "attributeValues" : {
        ## Add an item for each field you would like to store to Amazon DynamoDB. **
        "title" : $util.dynamodb.toDynamoDBJson($ctx.args.title),
        "content": $util.dynamodb.toDynamoDBJson($ctx.args.content),
        "owner": $util.dynamodb.toDynamoDBJson($context.identity.username)
    },
    "condition" : {
        "expression": $util.toJson("attribute_not_exists(id) AND $expression"),
        "expressionValues": $utils.toJson($expressionValues)
    }
}
```

ユースケース: グループが既存レコードを更新可能

```
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
    #set( $val = {})
    #set( $test = $val.put("S", $group))
    #set( $values = $expressionValues.put(":var$foreach.count", $val))
    #if ( $foreach.hasNext )
    #set( $expression = "${expression} OR" )
    #end
#end
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "update":{
                "expression" : "SET title = :title, content = :content",
        "expressionValues": {
            ":title" : $util.dynamodb.toDynamoDBJson($ctx.args.title),
```

```
            ":content" : $util.dynamodb.toDynamoDBJson($ctx.args.content)
        }
    },
    "condition" : {
        "expression": $util.toJson($expression),
        "expressionValues": $utils.toJson($expressionValues)
    }
}
```

パブリックレコードとプライベートレコード

条件フィルターを使用すると、データをプライベート、パブリック、またはその他のブール型チェックとしてマークすることもできます。それを認可フィルタの一部としてレスポンステンプレート内に組み込むことができます。このチェックを使用すると、グループメンバーシップを制御することなく、データを一時的に隠したり、ビューから除外したりできます。

たとえば、DynamoDB テーブル内の各項目に、yes または no のいずれかの値を持つ public という属性を追加するとします。次のレスポンステンプレートを GetItem 呼び出しで使用すると、アクセス権があるグループにユーザーが属していて、かつ、そのデータがパブリックとマークされている場合にのみ、データを表示できます。

```
#set($permissions = $context.result.GroupsCanAccess)
#set($claimPermissions = $context.identity.claims.get("cognito:groups"))

#foreach($per in $permissions)
    #foreach($cgroups in $claimPermissions)
        #if($cgroups == $per)
            #set($hasPermission = true)
        #end
    #end
#end

#if($hasPermission && $context.result.public == 'yes')
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

また、上記のコードで論理 OR (| |) を使用すると、ユーザーにレコードへのアクセス許可があるか、または、レコードがパブリックである場合に、そのユーザーに読み取りを許可できます。

```
#if($hasPermission || $context.result.public == 'yes')
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

通常、認可チェックを実行する際に、標準的な演算子 ==、!=、&&、および || が役立ちます。

リアルタイムデータ

クライアントがサブスクリプションを作成したときに、このドキュメントで前述したのと同じ手法で、きめ細かなアクセス制御コントロールを GraphQL のサブスクリプションに適用できます。サブスクリプションフィールドにリゾルバーをアタッチすると、そのポイントで、データソースからデータをクエリし、リクエストまたはレスポンスのいずれかのマッピングテンプレートで条件ロジックを実行できます。そのデータ構造が、GraphQL サブスクリプションで返される型と一致している限り、追加のデータ (サブスクリプションからの初期結果など) をクライアントに返すこともできます。

ユースケース: ユーザーが特定の対話のみのサブスクライブ可能

GraphQL サブスクリプションを使用したリアルタイムデータのよくあるユースケースは、メッセージングやプライベートチャットのアプリケーションを構築することです。複数のユーザーに対応したチャットアプリケーションを作成する場合、2 人または複数ユーザーの間で対話が行われます。ユーザーは、プライベートまたはパブリックの「ルーム」にグループ化されます。したがって、ユーザーにアクセス権が付与されている対話 (1 対 1 またはグループ間の) をサブスクライブする 1 人のユーザーのみを認可します。デモの目的で、以下の単純な例では、1 人のユーザーが別のユーザーにプライベートのメッセージを送信するユースケースを示します。次の 2 つの Amazon DynamoDB テーブルをセットアップします。

- Messages テーブル: (プライマリキー) `toUser`、(ソートキー) `id`
- Permissions テーブル: (プライマリキー) `username`

Messages テーブルには、GraphQL ミューテーション経由で送信される実際のメッセージが保存されます。Permissions テーブルは、クライアントの接続時に認可のために GraphQL サブスクリプションによってチェックされます。以下の例では、次の GraphQL スキーマを使用していることを前提としています。

```
input CreateUserPermissionsInput {
```

```
    user: String!
    isAuthorizedForSubscriptions: Boolean
}

type Message {
    id: ID
    toUser: String
    fromUser: String
    content: String
}

type MessageConnection {
    items: [Message]
    nextToken: String
}

type Mutation {
    sendMessage(toUser: String!, content: String!): Message
    createUserPermissions(input: CreateUserPermissionsInput!): UserPermissions
    updateUserPermissions(input: UpdateUserPermissionInput!): UserPermissions
}

type Query {
    getMyMessages(first: Int, after: String): MessageConnection
    getUserPermissions(user: String!): UserPermissions
}

type Subscription {
    newMessage(toUser: String!): Message
        @aws_subscribe(mutations: ["sendMessage"])
}

input UpdateUserPermissionInput {
    user: String!
    isAuthorizedForSubscriptions: Boolean
}

type UserPermissions {
    user: String
    isAuthorizedForSubscriptions: Boolean
}

schema {
    query: Query
```

```
    mutation: Mutation
    subscription: Subscription
}
```

以下では、サブスクリプションリゾルバーを示すために、一部の標準オペレーション (createUserPermissions() など) は取り上げていませんが、DynamoDB リゾルバーで標準実装されています。代わりに、リゾルバーでのサブスクリプションの認可フローを中心に説明します。ユーザー間でメッセージを送信するには、sendMessage() フィールドにリゾルバーをアタッチし、次のリクエストテンプレートを使用して Messages テーブルデータソースを選択します。

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "toUser" : $util.dynamodb.toDynamoDBJson($ctx.args.toUser),
        "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
    },
    "attributeValues" : {
        "fromUser" : $util.dynamodb.toDynamoDBJson($context.identity.username),
        "content" : $util.dynamodb.toDynamoDBJson($ctx.args.content),
    }
}
```

この例では、$context.identity.username を使用します。これにより、 AWS Identity and Access Management または Amazon Cognito ユーザーのユーザー情報が返されます。レスポンステンプレートは、$util.toJson($ctx.result) の単純なパススルーです。保存してスキーマページに戻ります。次に、newMessage()Permissions テーブルをデータソースとして使用し、次のリクエストマッピングテンプレートを使用して、 サブスクリプションにリゾルバーをアタッチします。

```
{
    "version": "2018-05-29",
    "operation": "GetItem",
    "key": {
        "username": $util.dynamodb.toDynamoDBJson($ctx.identity.username),
    },
}
```

次のレスポンスマッピングテンプレートを使用し、Permissions テーブルからのデータを使用して認可チェックを実行します。

```
#if(! ${context.result})
```

```
    $utils.unauthorized()
#elseif(${context.identity.username} != ${context.arguments.toUser})
    $utils.unauthorized()
#elseif(! ${context.result.isAuthorizedForSubscriptions})
    $utils.unauthorized()
#else
##User is authorized, but we return null to continue
    null
#end
```

この例の場合、3 つの認可チェックを実行しています。1 つ目は、結果が返されることを確認します。2 つ目は、そのユーザーが別のユーザーに対するメッセージをサブスクライブしていないことを確認しています。3 番目のチェックでは、isAuthorizedForSubscriptions として保存されている BOOL の DynamoDB 属性をチェックすることで、そのユーザーが任意のフィールドへのサブスクライブを許可されていることを確認しています。

テストするには、Amazon Cognito ユーザープールと「Nadia」という名前のユーザーを使用して AWS AppSync コンソールにサインインし、次の GraphQL サブスクリプションを実行します。

```
subscription AuthorizedSubscription {
    newMessage(toUser: "Nadia") {
        id
        toUser
        fromUser
        content
    }
}
```

Permissions テーブルに、Nadia の username キー属性に対して isAuthorizedForSubscriptions が true に設定されているレコードがある場合は、正常なレスポンスが表示されます。上記の username クエリで別の newMessage() を試行すると、エラーが返されます。

AWS WAF を使用して AWS AppSync APIsを保護する

AWS WAF は、ウェブアプリケーションと APIsから保護するのに役立つウェブアプリケーションファイアウォールです。お客様が定義するカスタマイズ可能なウェブセキュリティルールと条件に基づいて、ウェブリクエストを許可、ブロック、またはモニタリング (カウント) する一連のルール (ウェブアクセスコントロールリストまたはウェブ ACL と呼ばれます) を設定することができます。 AWS AppSync API を と統合すると AWS WAF、API が受け入れる HTTP トラフィックをより

詳細に制御し、可視化できます。詳細については AWS WAF、「 AWS WAF デベロッパーガイド」のAWS WAF 「 の仕組み」を参照してください。

を使用して AWS WAF 、SQL インジェクションやクロスサイトスクリプティング (XSS) 攻撃などの一般的なウェブエクスプロイトから AppSync API を保護できます。これらは、API の可用性とパフォーマンスに影響を与え、セキュリティを侵害したり、過剰なリソースを消費したりする可能性があります。例えば、指定した IP アドレス範囲からのリクエスト、CIDR ブロックからのリクエスト、特定の国またはリージョンからのリクエスト、悪意のある SQL コードを含むリクエスト、悪意のあるスクリプトを含むリクエストを許可またはブロックするルールを作成できます。

また、HTTP ヘッダー、メソッド、クエリ文字列、URI、およびリクエストボディの指定された文字列または定型表現パターン (最初の 8 KB に制限されます) に一致するルールを作成することもできます。さらに、特定のユーザーエージェント、悪質なボット、またはコンテンツスクレーパーからの攻撃をブロックするルールを作成できます。例えば、レートベースのルールを使用して、継続的に更新される後続の 5 分間で、各クライアント IP によって許可されるウェブリクエストの数を指定できます。

サポートされるルールのタイプと追加 AWS WAF 機能の詳細については、 AWS WAF デベロッパーガイドと AWS WAF API リファレンスを参照してください。

> **Important**
>
> AWS WAF は、ウェブエクスプロイトに対する防御の最前線です。API で AWS WAF が有効になっている場合は、API キー認可、IAM ポリシー、OIDC トークン、および Amazon Cognito ユーザープールなど他のアクセスコントロール機能の前に AWS WAF ルールが評価されます。

AppSync API を と統合する AWS WAF

Appsync API は、 AWS Management Console、 AWS CLI、またはその他の互換性のあるクライアント AWS WAF を使用して AWS CloudFormationと統合できます。

AWS AppSync API を と統合するには AWS WAF

1. AWS WAF ウェブ ACL を作成します。AWS WAF コンソールを使用する詳細なステップについては、「ウェブ ACL の作成」を参照してください。
2. ウェブ ACL のルールを定義します。1 つまたは複数のルールが、ウェブ ACL を作成するプロセスで定義されます。ルールを構成する方法の詳細については、AWS WAF ルールを参照してく

ださい。 AWS AppSync API に定義できる便利なルールの例については、「」を参照してくださいウェブ ACL のルールの作成。

3. ウェブ ACL を AWS AppSync API に関連付けます。このステップは、AWS WAF コンソールまたは、AppSync コンソールで実行できます。

   - ウェブ ACL を AWS WAF コンソールの AWS AppSync API に関連付けるには、「 デベロッパーガイド」の「ウェブ ACL と AWS リソースの関連付けまたは関連付け解除」の手順に従います。 AWS WAF
   - コンソールでウェブ ACL を AWS AppSync API AWS AppSync に関連付けるには

     a. にサインイン AWS Management Console し、AppSync コンソールを開きます。

     b. ウェブ ACL に関連付ける API を選択します。

     c. ナビゲーションペインで [設定] を選択します。

     d. Web アプリケーションファイアウォールセクションで、有効化 AWS WAFをオンにします。

     e. ウェブ ACLドロップダウンリストで、API に関連付けるウェブ ACL の名前を選択します。

     f. 保存を選択してウェブ ACL を API に関連付けます。

> **Note**
>
> AWS WAF コンソールでウェブ ACL を作成した後、新しいウェブ ACL が使用可能になるまでに数分かかることがあります。Web アプリケーションファイアウォールメニューで新しく作成されたウェブ ACL が表示されない場合、数分待ってから、ウェブ ACL を API に関連付ける手順を再試行します。

> **Note**
>
> AWS WAF 統合は、リアルタイムエンドポイントの `Subscription registration message`イベントのみをサポートします。 AWS AppSync は、`Subscription registration message`ブロックされた の`start_ack`メッセージではなく、エラーメッセージで応答します AWS WAF。

ウェブ ACL を AWS AppSync API に関連付けると、 AWS WAF APIs を使用してウェブ ACL を管理します。 AWS AppSync API を別のウェブ ACL に関連付ける場合を除き、ウェブ ACL を AWS AppSync API に再関連付けする必要はありません。

ウェブ ACL のルールの作成

ルールは、ウェブリクエストの検査方法と、ウェブリクエストが検査基準に一致した場合の処理を定義します。ルール自体は に存在しません AWS WAF 。ルールが定義されているルールグループまたはウェブ ACL に含まれる名前を使用してルールにアクセスします。詳細については、「AWS WAF ルール」を参照してください。次の例は、AppSync API の保護に役立つルールを定義し、関連付ける方法を示しています。

Example リクエストボディのサイズを制限するウェブ ACL ルール

次に、リクエストのボディサイズを制限するルールの例を示します。これは、 AWS WAF コンソールでウェブ ACL を作成するときにルール JSON エディタに入力されます。

```
{
    "Name": "BodySizeRule",
    "Priority": 1,
    "Action": {
        "Block": {}
    },
    "Statement": {
        "SizeConstraintStatement": {
            "ComparisonOperator": "GE",
            "FieldToMatch": {
                "Body": {}
            },
            "Size": 1024,
            "TextTransformations": [
                {
                    "Priority": 0,
                    "Type": "NONE"
                }
             ]
          }
       },
       "VisibilityConfig": {
           "CloudWatchMetricsEnabled": true,
           "MetricName": "BodySizeRule",
           "SampledRequestsEnabled": true
```

```
        }
}
```

前述のルール例を使用してウェブ ACL を作成したら、それを AppSync API に関連付ける必要があります。を使用する代わりに AWS Management Console、 AWS CLI 次のコマンドを実行して、 でこのステップを実行できます。

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

変更が伝播されるまで数分かかる場合がありますが、このコマンドを実行した後、1024 バイトを超える本文を含む要求は、 AWS AppSyncによって拒否されます。

> **Note**
>
> AWS WAF コンソールで新しいウェブ ACL を作成した後、ウェブ ACL が API に関連付けるまでに数分かかることがあります。CLI コマンドを実行して、`WAFUnavailableEntityException` エラーを取得し、数分待ってからコマンドを再実行してください。

Example 単一の IP アドレスからの要求を制限するウェブ ACL ルール

以下は、5 分間の単一の IP アドレスからの 100 リクエストに AppSync API をスロットリングするルールの例です。これは、 AWS WAF コンソールでレートベースのルールを使用してウェブ ACL を作成するときに、ルール JSON エディタに入力されます。

```
{
  "Name": "Throttle",
  "Priority": 0,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "Throttle"
  },
  "Statement": {
    "RateBasedStatement": {
      "Limit": 100,
```

```
      "AggregateKeyType": "IP"
    }
  }
}
```

前述のルール例を使用してウェブ ACL を作成したら、それを AppSync API に関連付ける必要があります。このステップは、次のコマンド AWS CLI を実行して で実行できます。

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

Example API に対するすべての GraphQL __schema イントロスペクションクエリを防止するためのウェブ ACL ルール

次に、API に対するすべての G GraphQL __schema イントロスペクションクエリを禁止するルールの例を示します。文字列「__schema」を含む HTTP ボディはすべてブロックされます。これは、AWS WAF コンソールでウェブ ACL を作成するときにルール JSON エディタに入力されます。

```
{
  "Name": "BodyRule",
  "Priority": 5,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "BodyRule"
  },
  "Statement": {
    "ByteMatchStatement": {
      "FieldToMatch": {
        "Body": {}
      },
      "PositionalConstraint": "CONTAINS",
      "SearchString": "__schema",
      "TextTransformations": [
        {
          "Type": "NONE",
          "Priority": 0
        }
      ]
    }
```

```
  }
}
```

前述のルール例を使用してウェブ ACL を作成したら、それを AppSync API に関連付ける必要があります。このステップは、次のコマンド AWS CLI を実行して で実行できます。

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

AWS AppSync のセキュリティ

のクラウドセキュリティが最優先事項 AWS です。お客様は AWS 、セキュリティを最も重視する組織の要件を満たすように構築されたデータセンターとネットワークアーキテクチャからメリットを得られます。

セキュリティは、 AWS とユーザーの間で共有される責任です。責任共有モデルでは、これをクラウドのセキュリティおよびクラウド内のセキュリティと説明しています。

- クラウドのセキュリティ – 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 AWS 責任があります。 AWS また、 では、安全に使用できるサービスも提供しています。サードパーティーの監査者は、AWS コンプライアンスプログラムコンプライアンスプログラムの一環として、当社のセキュリティの有効性を定期的にテストおよび検証。 AWS AppSync 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AWS による対象範囲内のサービスコンプライアンスプログラム」を参照してください。
- クラウド内のセキュリティ – お客様の責任は、使用する AWS サービスによって決まります。また、ユーザーは、データの機密性、会社の要件、適用される法律や規制など、その他の要因についても責任を負います。

このドキュメントは、 AWS AppSync を使用する際の責任共有モデルの適用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ように AWS AppSync を設定する方法について説明します。また、 AWS AppSync 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 AppSync でのデータ保護
- AWS AppSync のコンプライアンス検証
- AWS AppSync のインフラストラクチャセキュリティ
- AWS AppSync の耐障害性
- AWS AppSync の Identity and Access Management
- を使用した Logging AWS AppSync API コール AWS CloudTrail
- のセキュリティのベストプラクティス AWS AppSync

AWS AppSync でのデータ保護

責任 AWS 共有モデル、 AWS AppSync でのデータ保護に適用されます。このモデルで説明されているように、 AWS はすべての 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 AWS クラウド。ユーザーは、このインフラストラクチャでホストされ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に関する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。

データ保護の目的で、認証情報を保護し AWS アカウント 、 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設定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が各ユーザーに付与されます。また、次の方法でデータを保護することもお勧めします:

- 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、TLS 1.3 をお勧めします。
- で API とユーザーアクティビティのログ記録を設定します AWS CloudTrail。CloudTrail 証跡を使用して AWS アクティビティをキャプチャする方法については、「 AWS CloudTrail ユーザーガイド」のCloudTrail 証跡の使用」を参照してください。
- AWS 暗号化ソリューションと、 内のすべてのデフォルトのセキュリティコントロールを使用します AWS のサービス。
- Amazon Macie などの高度な管理された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 AWS を介して にアクセスするときに FIPS 140-3 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) 140-3」を参照してください。

お客様の E メールアドレスなどの極秘または機密情報を、タグ、または [名前] フィールドなどの自由形式のテキストフィールドに含めないことを強くお勧めします。これは、コンソール、API、または SDK を使用して AWS AppSync AWS CLIまたは他の AWS のサービス を操作する場合も同様です。 AWS SDKs タグ、または名前に使用される自由記述のテキストフィールドに入力したデータは、請求または診断ログに使用される場合があります。外部サーバーに URL を提供する場合、そのサーバーへのリクエストを検証できるように、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

AWS AppSyncは、すべての AWS サービスと同様に、 AWS 公開された APIsと SDKs を使用する際の通信に TLS1.2 以降を使用します。

Amazon DynamoDB などの他の AWS サービス AWS AppSync で を使用すると、転送中の暗号化が保証されます。特に指定がない限り、すべての AWS サービスは TLS 1.2 以降を使用して相互に通信します。Amazon EC2 または CloudFront を使用するリゾルバーの場合、TLS (HTTPS) が設定され、安全であることを確認するのはユーザーの責任です。Amazon EC2 での HTTPS の設定については、Amazon EC2 ユーザーガイドの「Amazon Linux 2 での SSL/TLS の設定」を参照してください。CloudFront での HTTPS の設定については、CloudFront ユーザーガイドの「Amazon CloudFront のHTTPS」を参照してください。

AWS AppSync のコンプライアンス検証

サードパーティーの監査者は、複数のコンプライアンスプログラムの一環として、 AWS AppSync のセキュリティと AWS コンプライアンスを評価します。 AWS AppSync は、SOC、PCI、HIPAA/HIPAA BAA、IRAP、C5、ENS High、OSPAR、および HITRUST CSF プログラムに準拠しています。

AWS のサービス が特定のコンプライアンスプログラムの範囲内にあるかどうかを確認するには、「コンプライアンスAWS のサービス プログラムによる対象範囲内」を参照して、関心のあるコンプライアンスプログラムを選択します。一般的な情報については、AWS 「 Compliance ProgramsAssurance」を参照してください。

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 AWS Artifact。詳細については、「Downloading Reports in AWS Artifact」を参照してください。

を使用する際のお客様のコンプライアンス責任 AWS のサービス は、お客様のデータの機密性、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。 は、コンプライアンスに役立つ以下のリソース AWS を提供します。

- セキュリティのコンプライアンスとガバナンス – これらのソリューション実装ガイドでは、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の機能をデプロイする手順を示します。
- HIPAA 対応サービスのリファレンス – HIPAA 対応サービスの一覧が提供されています。すべてが HIPAA AWS のサービス の対象となるわけではありません。
- AWS コンプライアンスリソース – このワークブックとガイドのコレクションは、お客様の業界や地域に適用される場合があります。

- AWS カスタマーコンプライアンスガイド – コンプライアンスの観点から責任共有モデルを理解します。このガイドは、複数のフレームワーク (米国国立標準技術研究所 (NIST)、Payment Card Industry Security Standards Council (PCI)、国際標準化機構 (ISO) を含む) のセキュリティコントロールを保護し、そのガイダンスに AWS のサービス マッピングするためのベストプラクティスをまとめたものです。
- 「 デベロッパーガイド」の「ルールによるリソースの評価」 – この AWS Config サービスは、リソース設定が内部プラクティス、業界ガイドライン、および規制にどの程度準拠しているかを評価します。 AWS Config
- AWS Security Hub – これにより AWS のサービス 、 内のセキュリティ状態を包括的に把握できます AWS。Security Hub では、セキュリティコントロールを使用して AWS リソースを評価し、セキュリティ業界標準とベストプラクティスに対するコンプライアンスをチェックします。サポートされているサービスとコントロールの一覧については、Security Hub のコントロールリファレンスを参照してください。
- Amazon GuardDuty – 不審なアクティビティや悪意のあるアクティビティがないか環境をモニタリングすることで AWS アカウント、、ワークロード、コンテナ、データに対する潜在的な脅威 AWS のサービス を検出します。GuardDuty を使用すると、特定のコンプライアンスフレームワークで義務付けられている侵入検知要件を満たすことで、PCI DSS などのさまざまなコンプライアンス要件に対応できます。
- AWS Audit Manager – これにより AWS のサービス 、 AWS 使用状況を継続的に監査し、リスクの管理方法と規制や業界標準への準拠を簡素化できます。

AWS AppSync 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 AppSync 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 がインフラストラクチャ AWS 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を使用して AWS 環境を設計するには、「Security Pillar AWS Well-Architected Framework」の「Infrastructure Protection」を参照してください。

AWS 公開された API コールを使用して、ネットワーク経由で AWS AppSync 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 が必須で、TLS 1.3 をお勧めします。
- 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、ほとんどの最新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、一時的なセキュリティ認証情報を生成し、リクエストに署名することもできます。

AWS AppSync の耐障害性

AWS 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ています。 AWS リージョンは、低レイテンシー、高スループット、高度に冗長なネットワークで接続された複数の物理的に分離されたアベイラビリティーゾーンを提供します。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が高く、フォールトトレラントで、スケーラブルです。

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。

AWS グローバルインフラストラクチャに加えて、 AWS AppSync ではほとんどのリソースを AWS CloudFormation テンプレートを使用して定義できます。 AWS CloudFormation テンプレートを使用して AWS AppSync リソースを宣言する例については、 AWS ブログのAWS AppSync 「パイプラインリゾルバーの実用的なユースケース」およびAWS CloudFormation 「 ユーザーガイド」を参照してください。

AWS AppSync の Identity and Access Management

AWS Identity and Access Management (IAM) は、管理者が AWS リソースへのアクセスを安全に制御 AWS のサービス するのに役立つ です。IAM 管理者は、誰を認証 (サインイン) し、誰に AWS AppSync リソースの使用を許可する (アクセス許可を付与する) かを制御します。IAM は、追加料金なしで使用できる AWS のサービス です。

トピック

- 対象者
- アイデンティティを使用した認証
- ポリシーを使用したアクセスの管理
- IAM での AWS AppSync の仕組み
- AWS AppSync のアイデンティティベースのポリシー

- AWS AppSync 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

対象者

AWS Identity and Access Management (IAM) の使用方法は、 AWS AppSync で行う作業によって異なります。

サービスユーザー – AWS AppSync サービスを使用してジョブを実行する場合、管理者は必要な認証情報とアクセス許可を提供します。さらに多くの AWS AppSync 機能を使用して作業を行う場合は、追加のアクセス許可が必要になる場合があります。アクセスの管理方法を理解すると、管理者に適切なアクセス許可をリクエストするのに役に立ちます。 AWS AppSync の機能にアクセスできない場合は、「AWS AppSync 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 AWS AppSync リソースを担当している場合は、通常、 AWS AppSync へのフルアクセスがあります。サービスユーザーがどの AWS AppSync 機能と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。会社で AWS AppSync で IAM を使用する方法の詳細については、「」を参照してくださいIAM での AWS AppSync の仕組み。

IAM 管理者 – IAM 管理者は、 AWS AppSync へのアクセスを管理するポリシーの作成方法の詳細について確認する場合があります。IAM で使用できる AWS AppSync アイデンティティベースのポリシーの例を表示するには、「」を参照してくださいAWS AppSync のアイデンティティベースのポリシー。

アイデンティティを使用した認証

認証は、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 ロールを引き受けることによって、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 認証情報は、フェデレーション 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。

ユーザーの種類に応じて、 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、「 AWS サインイン ユーザーガイド」の「 へのサインイン AWS アカウント方法」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、 はソフトウェア開発キット (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を暗号化して署名します。AWS ツールを使用しない場合は、自分でリクエストに署名する必要があります。リクエストに自分で署名する推奨方法の使用については、「IAM ユーザーガイド」の「API リクエストに対するAWS Signature Version 4」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。たとえば、 では、アカウントのセキュリティを高めるために多要素認証 (MFA) を使用する AWS 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「多要素認証」および「IAM ユーザーガイド」の「IAM のAWS 多要素認証」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。

フェデレーティッドアイデンティティ

ベストプラクティスとして、管理者アクセスを必要とするユーザーを含む人間のユーザーに、ID プロバイダーとのフェデレーションを使用して一時的な認証情報 AWS のサービス を使用して にアクセスすることを要求します。

フェデレーティッド ID は、エンタープライズユーザーディレクトリ、ウェブ ID プロバイダー、AWS Directory Service、アイデンティティセンターディレクトリ、または ID ソースを介して提供された認証情報 AWS のサービス を使用して にアクセスするユーザーです。フェデレーティッドアイデンティティがアクセスすると AWS アカウント、ロールを引き受け、ロールは一時的な認証情報を提供します。

アクセスを一元管理する場合は、 AWS IAM Identity Centerを使用することをお勧めします。IAM Identity Center でユーザーとグループを作成するか、独自の ID ソースのユーザーとグループのセッ

トに接続して同期し、すべての AWS アカウント とアプリケーションで使用できます。IAM Identity Center の詳細については、「AWS IAM Identity Center ユーザーガイド」の「What is IAM Identity Center?」(IAM Identity Center とは) 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 ID 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 ユーザーに関するユースケース」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 ID 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。で IAM ロールを一時的に引き受けるには AWS Management Console、ユーザーから IAM ロール (コンソール) に切り替えることができます。ロールを引き受けるには、 または AWS API オペレーションを AWS CLI 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「ロールを引き受けるための各種方法」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ロール

については、「IAM ユーザーガイド」の「サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する」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 User Guide」の「Permission sets」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、 (プロキシとしてロールを使用する代わりに) リソースに直接ポリシーをアタッチできます。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アクションを実行すると AWS、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 へのリクエストをリクエストする 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「AWS のサービスに許可を委任するロールを作成する」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、サービスリンク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。EC2 インスタンスに AWS ロールを割り当て、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。

ポリシーを使用したアクセスの管理

でアクセスを制御する AWS には、ポリシーを作成し、ID AWS またはリソースにアタッチします。ポリシーは AWS 、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) 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 JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

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受けることができます。

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

ティティベースポリシーの作成方法については、「IAM ユーザーガイド」の「カスタマー管理ポリシーでカスタム IAM アクセス許可を定義する」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「管理ポリシーとインラインポリシーのいずれかを選択する」を参照してください。

リソースベースのポリシー

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには例として、IAM ロールの信頼ポリシーや Amazon S3 バケットポリシーがあげられます。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。ポリシーがアタッチされているリソースの場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーでは、プリンシパルを指定する必要があります。プリンシパルには、アカウント、ユーザー、ロール、フェデレーティッドユーザー、または を含めることができます AWS のサービス。

リソースベースのポリシーは、そのサービス内にあるインラインポリシーです。リソースベースのポリシーでは、IAM の AWS マネージドポリシーを使用できません。

アクセスコントロールリスト (ACL)

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするための許可を持つかを制御します。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

Amazon S3、および Amazon VPC は AWS WAF、ACLs。ACL の詳細については、「Amazon Simple Storage Service デベロッパーガイド」の「アクセスコントロールリスト (ACL) の概要」を参照してください。

その他のポリシータイプ

AWS は、一般的で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) の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、ビジネスが所有する複数の AWS アカウント をグループ化して一元管理するためのサービスです。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「サービスコントロールポリシー (SCP)」を参照してください。
- リソースコントロールポリシー (RCP) – RCP は、所有する各リソースにアタッチされた IAM ポリシーを更新することなく、アカウント内のリソースに利用可能な最大数のアクセス許可を設定するために使用できる JSON ポリシーです。RCP は、メンバーアカウントのリソースのアクセス許可を制限し、組織に属しているかどうかにかかわらず AWS アカウントのルートユーザー、 を含む ID の有効なアクセス許可に影響を与える可能性があります。RCP をサポートする のリストを含む Organizations と RCP の詳細については、AWS Organizations RCPs「リソースコントロールポリシー (RCPs」を参照してください。 AWS のサービス
- 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。が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。

IAM での AWS AppSync の仕組み

IAM を使用して AWS AppSync へのアクセスを管理する前に、 AWS AppSync で使用できる IAM 機能を確認してください。

AWS AppSync で使用できる IAM 機能

IAM の機能	AWS AppSync のサポート
アイデンティティベースポリシー	はい
リソースベースのポリシー	いいえ
ポリシーアクション	はい
ポリシーリソース	あり
ポリシー条件キー	いいえ
ACL	いいえ
ABAC (ポリシー内のタグ)	部分的
一時的な認証情報	あり
転送アクセスセッション (FAS)	部分的
サービスロール	いいえ
サービスリンクロール	部分的

AWS AppSync およびその他の AWS のサービスがほとんどの IAM 機能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。

AWS AppSync の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のポリシーのサポート: あり

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。ID ベースのポリシーの作成方法については、「IAM ユーザーガイド」の「カスタマー管理ポリシーでカスタム IAM アクセス許可を定義する」を参照してください。

IAM アイデンティティベースのポリシーでは、許可または拒否するアクションとリソース、およびアクションを許可または拒否する条件を指定できます。プリンシパルは、それが添付されているユーザーまたはロールに適用されるため、アイデンティティベースのポリシーでは指定できません。JSON ポリシーで使用できるすべての要素について学ぶには、「IAM ユーザーガイド」の「IAM JSON ポリシーの要素のリファレンス」を参照してください。

AWS AppSync のアイデンティティベースのポリシーの例

AWS AppSync アイデンティティベースのポリシーの例を表示するには、「」を参照してくださいAWS AppSync のアイデンティティベースのポリシー。

AWS AppSync 内のリソースベースのポリシー

リソースベースのポリシーのサポート: なし

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには例として、IAM ロールの信頼ポリシーや Amazon S3 バケットポリシーがあげられます。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。ポリシーがアタッチされているリソースの場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーでは、プリンシパルを指定する必要があります。プリンシパルには、アカウント、ユーザー、ロール、フェデレーティッドユーザー、または を含めることができます AWS のサービス。

クロスアカウントアクセスを有効にするには、アカウント全体、または別のアカウントの IAM エンティティをリソースベースのポリシーのプリンシパルとして指定します。リソースベースのポリシーにクロスアカウントのプリンシパルを追加しても、信頼関係は半分しか確立されない点に注意してください。プリンシパルとリソースが異なる場合 AWS アカウント、信頼されたアカウントの IAM 管理者は、プリンシパルエンティティ (ユーザーまたはロール) にリソースへのアクセス許可も付与する必要があります。IAM 管理者は、アイデンティティベースのポリシーをエンティティにアタッチすることで権限を付与します。ただし、リソースベースのポリシーで、同じアカウントのプリンシパルへのアクセス権が付与されている場合は、アイデンティティベースのポリシーをさらに付与する必要はありません。詳細については、「IAM ユーザーガイド」の「IAM でのクロスアカウントリソースアクセス」を参照してください。

AWS AppSync のポリシーアクション

ポリシーアクションのサポート:あり

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

JSON ポリシーの `Action` 要素にはポリシー内のアクセスを許可または拒否するために使用できるアクションが記述されます。ポリシーアクションの名前は通常、関連付けられた 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。

このアクションは関連付けられたオペレーションを実行するためのアクセス許可を付与するポリシーで使用されます。

AWS AppSync アクションのリストを確認するには、「サービス認可リファレンス」のAWS AppSync で定義されるアクション」を参照してください。

AWS AppSync のポリシーアクションは、アクションの前に次のプレフィックスを使用します。

```
appsync
```

単一のステートメントで複数のアクションを指定するには、アクションをカンマで区切ります。

```
"Action": [
      "appsync:action1",
      "appsync:action2"
         ]
```

AWS AppSync アイデンティティベースのポリシーの例を表示するには、「」を参照してくださいAWS AppSync のアイデンティティベースのポリシー。

AWS AppSync のポリシーリソース

ポリシーリソースのサポート: あり

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

Resource JSON ポリシー要素は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": "*"
```

AWS AppSync リソースタイプとその ARNsAWS AppSync で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、AWS AppSync で定義されるアクション」を参照してください。

AWS AppSync アイデンティティベースのポリシーの例を表示するには、「」を参照してくださいAWS AppSync のアイデンティティベースのポリシー。

AWS AppSync のポリシー条件キー

サービス固有のポリシー条件キーへのサポート: なし

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

Condition 要素 (または Condition ブロック) を使用すると、ステートメントが有効な条件を指定できます。Condition 要素はオプションです。イコールや未満などの 条件演算子 を使用して条件式を作成して、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ています。すべての AWS グローバル条件キーを確認するには、「IAM ユーザーガイド」のAWS 「グローバル条件コンテキストキー」を参照してください。

AWS AppSync 条件キーのリストを確認するには、「サービス認可リファレンス」の AWS AppSync 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、AWS AppSync で定義されるアクション」を参照してください。

AWS AppSync アイデンティティベースのポリシーの例を表示するには、「」を参照してくださいAWS AppSync のアイデンティティベースのポリシー。

AWS AppSync のアクセスコントロールリスト (ACL)

ACL のサポート: なし

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするための許可を持つかを制御します。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

AWS AppSync を使用した属性ベースのアクセスコントロール (ABAC)

ABAC (ポリシー内のタグ) のサポート: 一部

属性ベースのアクセス制御 (ABAC) は、属性に基づいてアクセス許可を定義する認可戦略です。ではAWS、これらの属性はタグと呼ばれます。タグは、IAM エンティティ (ユーザーまたはロール) および多くの AWS リソースにアタッチできます。エンティティとリソースのタグ付けは、ABAC の最初の手順です。その後、プリンシパルのタグがアクセスしようとしているリソースのタグと一致した場合にオペレーションを許可するように ABAC ポリシーをします。

ABAC は、急成長する環境やポリシー管理が煩雑になる状況で役立ちます。

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。

サービスがすべてのリソースタイプに対して 3 つの条件キーすべてをサポートする場合、そのサービスの値はありです。サービスが一部のリソースタイプに対してのみ 3 つの条件キーのすべてをサポートする場合、値は「部分的」になります。

ABAC の詳細については、「IAM ユーザーガイド」の「ABAC 認可でアクセス許可を定義する」を参照してください。ABAC をセットアップする手順を説明するチュートリアルについては、「IAM ユーザーガイド」の「属性ベースのアクセスコントロール (ABAC) を使用する」を参照してください。

AWS AppSync での一時的な認証情報の使用

一時的な認証情報のサポート: あり

一部の AWS のサービス は、一時的な認証情報を使用してサインインすると機能しません。一時的な認証情報 AWS のサービス を使用する などの詳細については、IAM ユーザーガイドの「IAM AWS のサービス と連携する 」を参照してください。

ユーザー名とパスワード以外の AWS Management Console 方法で にサインインする場合、一時的な認証情報を使用します。たとえば、会社のシングルサインオン (SSO) リンク AWS を使用して にアクセスすると、そのプロセスによって一時的な認証情報が自動的に作成されます。また、ユーザーとしてコンソールにサインインしてからロールを切り替える場合も、一時的な認証情報が自動的に作成されます。ロールの切り替えに関する詳細については、「IAM ユーザーガイド」の「ユーザーから IAM ロールに切り替える (コンソール)」を参照してください。

一時的な認証情報は、 AWS CLI または AWS API を使用して手動で作成できます。その後、これらの一時的な認証情報を使用してアクセスすることができます AWS。長期的なアクセスキーを使用する代わりに、一時的な認証情報を動的に生成 AWS することをお勧めします。詳細については、「IAM の一時的セキュリティ認証情報」を参照してください。

AWS AppSync の転送アクセスセッション

転送アクセスセッション (FAS) のサポート: 一部

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と AWS、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 へのリクエストをリクエストする 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。

AWS AppSync のサービスロール

サービスロールのサポート: なし

サービスロールとは、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 のサービスに許可を委任するロールを作成する」を参照してください。

> **Warning**
>
> サービスロールのアクセス許可を変更すると、 AWS AppSync 機能が破損する可能性があります。 AWS AppSync が指示する場合にのみ、サービスロールを編集します。

AWS AppSync の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をサポート： 一部

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、サービスリンク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。

サービスリンクロールの作成または管理の詳細については、「IAM ユーザーガイド」の「IAM と提携するAWS サービス」を参照してください。表の中から、[サービスにリンクされたロール] 列に Yes と記載されたサービスを見つけます。サービスにリンクされたロールに関するドキュメントをサービスで表示するには、[はい] リンクを選択します。

AWS AppSync のアイデンティティベースのポリシー

デフォルトでは、ユーザーとロールには、 AWS AppSync リソースを作成または変更するアクセス許可はありません。また、、 AWS Command Line Interface （ AWS CLI ） AWS Management Console、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、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。

これらサンプルの JSON ポリシードキュメントを使用して、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「IAM ポリシーを作成する (コンソール)」を参照してください。

各リソースタイプの ARN の形式など、 AWS AppSync で定義されるアクションとリソースタイプの詳細については、「サービス認可リファレンス」のAWS AppSync のアクション、リソース、および条件キー」を参照してください。 ARNs

IAM アイデンティティベースのポリシーを作成および設定するためのベストプラクティスについては、「the section called “IAM ポリシーのベストプラクティス”」を参照してください。

AWS AppSync の IAM アイデンティティベースのポリシーのリストについては、「」を参照してくださいAWSAWS AppSync の マネージドポリシー。

トピック

- AWS AppSync コンソールの使用
- 自分の権限の表示をユーザーに許可する
- 1 つの Amazon S3 バケットへのアクセス
- タグに基づく AWS AppSync widget の表示
- AWSAWS AppSync の マネージドポリシー

AWS AppSync コンソールの使用

AWS AppSync コンソールにアクセスするには、最小限のアクセス許可のセットが必要です。これらのアクセス許可により、 の AWS AppSync リソースの詳細を一覧表示および表示できます AWS アカウント。最小限必要な許可よりも制限が厳しいアイデンティティベースのポリシーを作成すると、そのポリシーを持つエンティティ (ユーザーまたはロール) に対してコンソールが意図したとおりに機能しません。

AWS CLI または AWS API のみを呼び出すユーザーには、最小限のコンソールアクセス許可を付与する必要はありません。代わりに、実行しようとしている API オペレーションに一致するアクションのみへのアクセスが許可されます。

IAM ユーザーとロールが引き続き AWS AppSync コンソールを使用できるようにするには、エンティティに AWS AppSync `ConsoleAccess`または `ReadOnly` AWS 管理ポリシーもアタッチします。詳細については、「IAM ユーザーガイド」の「ユーザーへのアクセス許可の追加」を参照してください。

自分の権限の表示をユーザーに許可する

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 Amazon S3 バケットへのアクセス

この例では、 AWS アカウントの IAM ユーザーに Amazon S3 バケットの 1 つへのアクセス権を付与しますexamplebucket。また、ユーザーがオブジェクトを追加、更新、および削除できるようにします。

このポリシーでは、ユーザーに s3:PutObject、s3:GetObject、s3:DeleteObject のアクセス許可を付与するだけでなく、s3:ListAllMyBuckets、s3:GetBucketLocation、および s3:ListBucket のアクセス許可も付与します。これらが、コンソールで必要とされる追加のアクセス許可です。またコンソール内のオブジェクトのコピー、カット、貼り付けを行うためには、s3:PutObjectAcl および s3:GetObjectAcl アクションが必要となります。コンソールを使用して、ユーザーにアクセス許可を付与し、テストする例の解説については、「チュートリアル例: ユーザーポリシーを使用したバケットへのアクセスのコントロール」を参照してください。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"ListBucketsInConsole",
         "Effect":"Allow",
         "Action":[
            "s3:ListAllMyBuckets"
         ],
         "Resource":"arn:aws:s3:::*"
      },
      {
         "Sid":"ViewSpecificBucketInfo",
         "Effect":"Allow",
         "Action":[
            "s3:ListBucket",
            "s3:GetBucketLocation"
         ],
         "Resource":"arn:aws:s3:::examplebucket"
      },
      {
         "Sid":"ManageBucketContents",
         "Effect":"Allow",
         "Action":[
            "s3:PutObject",
```

```
            "s3:PutObjectAcl",
            "s3:GetObject",
            "s3:GetObjectAcl",
            "s3:DeleteObject"
         ],
         "Resource":"arn:aws:s3:::examplebucket/*"
      }
   ]
}
```

タグに基づく AWS AppSync *widget* の表示

アイデンティティベースのポリシーの条件を使用して、タグに基づいて AWS AppSync リソースへのアクセスを制御できます。この例では、######を表示できるポリシーを作成する方法を示します。ただし、アクセス許可は、*widget* タグ Owner にそのユーザーのユーザー名の値がある場合にのみ付与されます。このポリシーでは、このアクションをコンソールで実行するために必要なアクセス許可も付与します。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ListWidgetsInConsole",
            "Effect": "Allow",
            "Action": ":ListWidgets",
            "Resource": "*"
        },
        {
            "Sid": "ViewWidgetIfOwner",
            "Effect": "Allow",
            "Action": ":GetWidget",
            "Resource": "arn:aws::*:*:widget/*",
            "Condition": {
                "StringEquals": {"aws:ResourceTag/Owner": "${aws:username}"}
            }
        }
    ]
}
```

このポリシーをアカウントの IAM ユーザーにアタッチできます。という名前のユーザーが AWS AppSync ######を表示richard-roeしようとする場合、######には 0wner=richard-roeまたは のタグを付ける必要がありますowner=richard-roe。それ以外の場合、アクセスは拒否されます。条件キー名では大文字と小文字が区別されないため、条件タグキー 0wner は 0wner と owner の両方に一致します。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。

AWSAWS AppSync の マネージドポリシー

ユーザー、グループ、ロールにアクセス許可を追加するには、自分でポリシーを記述するよりも AWS 管理ポリシーを使用する方が簡単です。チームに必要なアクセスのみを許可する IAM のカスタマー管理のポリシーを作成するには、時間と専門知識が必要です。すぐに開始するには、 AWS マネージドポリシーを使用できます。これらのポリシーは、一般的なユースケースをターゲット範囲に含めており、 AWS アカウントで利用できます。 AWS 管理ポリシーの詳細については、IAM ユーザーガイドの「 AWS 管理ポリシー」を参照してください。

AWS サービスは、 AWS 管理ポリシーを維持および更新します。 AWS 管理ポリシーのアクセス許可は変更できません。サービスは、新機能をサポートするために、 AWS 管理ポリシーに追加のアクセス許可を追加することがあります。この種類の更新はポリシーがアタッチされている、すべてのアイデンティティ (ユーザー、グループおよびロール) に影響を与えます。サービスは、新機能が起動されたとき、または新しいオペレーションが利用可能になったときに、 AWS マネージドポリシーを更新する可能性が最も高いです。サービスは AWS マネージドポリシーからアクセス許可を削除しないため、ポリシーの更新によって既存のアクセス許可が破損することはありません。

さらに、 は、複数のサービスにまたがるジョブ関数の マネージドポリシー AWS をサポートしています。例えば、ReadOnlyAccess AWS 管理ポリシーは、すべての AWS サービスとリソースへの読み取り専用アクセスを提供します。サービスが新機能を起動すると、 は新しいオペレーションとリソースの読み取り専用アクセス許可 AWS を追加します。ジョブ機能のポリシーの一覧および詳細については、「IAM ユーザーガイド」の「AWS のジョブ機能のマネージドポリシー」を参照してください。

AWS マネージドポリシー: AWSAppSyncInvokeFullAccess

AWSAppSyncInvokeFullAccess AWS 管理ポリシーを使用して、管理者が コンソールまたは個別に AWS AppSync サービスにアクセスできるようにします。

AWSAppSyncInvokeFullAccess ポリシーを IAM アイデンティティにアタッチできます。

アクセス許可の詳細

このポリシーには、以下のアクセス許可が含まれています。

- AWS AppSync – AWS AppSync 内のすべてのリソースへの完全な管理アクセスを許可します

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "appsync:GraphQL",
                "appsync:GetGraphqlApi",
                "appsync:ListGraphqlApis",
                "appsync:ListApiKeys"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS マネージドポリシー: AWSAppSyncSchemaAuthor

AWSAppSyncSchemaAuthor AWS マネージドポリシーを使用して、IAM ユーザーが にアクセスして GraphQL スキーマを作成、更新、クエリできるようにします。これらの権限でユーザーが実行できる内容については、「を使用した GraphQL APIsの設計 AWS AppSync」を参照してください。

AWSAppSyncSchemaAuthor ポリシーを IAM アイデンティティにアタッチできます。

アクセス許可の詳細

このポリシーには、以下のアクセス許可が含まれています。

- AWS AppSync – 以下のアクションを許可します。
  - GraphQL スキーマを作成する
  - GraphQL のタイプ、リゾルバー、関数の作成、変更、削除を許可する
  - リクエストとレスポンスのテンプレートのロジックを評価する
  - ランタイムとコンテキストを使用してコードを評価する
  - GraphQL クエリを GraphQL API に送信する
  - GraphQL データを取得する

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "appsync:GraphQL",
                "appsync:CreateResolver",
                "appsync:CreateType",
                "appsync:DeleteResolver",
                "appsync:DeleteType",
                "appsync:GetResolver",
                "appsync:GetType",
                "appsync:GetDataSource",
                "appsync:GetSchemaCreationStatus",
                "appsync:GetIntrospectionSchema",
                "appsync:GetGraphqlApi",
                "appsync:ListTypes",
                "appsync:ListApiKeys",
                "appsync:ListResolvers",
                "appsync:ListDataSources",
                "appsync:ListGraphqlApis",
                "appsync:StartSchemaCreation",
                "appsync:UpdateResolver",
                "appsync:UpdateType",
                "appsync:TagResource",
                "appsync:UntagResource",
                "appsync:ListTagsForResource",
```

```
                "appsync:CreateFunction",
                "appsync:UpdateFunction",
                "appsync:GetFunction",
                "appsync:DeleteFunction",
                "appsync:ListFunctions",
                "appsync:ListResolversByFunction",
                "appsync:EvaluateMappingTemplate",
                "appsync:EvaluateCode"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS マネージドポリシー: AWSAppSyncPushToCloudWatchLogs

AWS AppSync は Amazon CloudWatch を使用して、GraphQL リクエストのトラブルシューティングと最適化に使用できるログを生成することで、アプリケーションのパフォーマンスをモニタリングします。詳細については、「CloudWatch を使用した GraphQL API データのモニタリングと口グ」を参照してください。

AWSAppSyncPushToCloudWatchLogs AWS マネージドポリシーを使用して、 AWS AppSync が IAM ユーザーの CloudWatch アカウントにログをプッシュできるようにします。

AWSAppSyncPushToCloudWatchLogs ポリシーを IAM アイデンティティにアタッチできます。

アクセス許可の詳細

このポリシーには、以下のアクセス許可が含まれています。

- CloudWatch Logs – 指定した名前のロググループとストリームの作成を AWS AppSync に許可します。 AWS AppSync は、指定したログストリームにログイベントをプッシュします。

JSON

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS マネージドポリシー: AWSAppSyncAdministrator

AWSAppSyncAdministrator AWS 管理ポリシーを使用して、管理者が AWS コンソールを除くすべての AWS AppSync にアクセスできるようにします。

IAM エンティティに AWSAppSyncAdministrator をアタッチできます。 AWS AppSync は、ユーザーに代わってアクションを実行することを許可するサービスロールにもこのポリシーをアタッチします。

アクセス許可の詳細

このポリシーには、以下のアクセス許可が含まれています。

- AWS AppSync — AWS AppSync のすべてのリソースへの完全な管理アクセスを許可します
- IAM – 以下のアクションを許可します。
  - サービスにリンクされたロールを作成して、 AWS AppSync がユーザーに代わって他のサービスのリソースを分析できるようにします。
  - サービスにリンクされたロールの削除
  - サービスにリンクされたロールを他の AWS サービスに渡し、後でロールを引き受け、ユーザーに代わってアクションを実行する

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "appsync:*"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": [
                        "appsync.amazonaws.com"
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "appsync.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:DeleteServiceLinkedRole",
                "iam:GetServiceLinkedRoleDeletionStatus"
            ],
```

```
            "Resource": "arn:aws:iam::*:role/aws-service-role/
appsync.amazonaws.com/AWSServiceRoleForAppSync*"
        }
    ]
}
```

AWS マネージドポリシー: AWSAppSyncServiceRolePolicy

AWSAppSyncServiceRolePolicy AWS マネージドポリシーを使用して、 AWS AppSync が使用または管理する AWS サービスとリソースへのアクセスを許可します。

IAM エンティティに AWSAppSyncServiceRolePolicy をアタッチすることはできません。このポリシーは、 AWS AppSync がユーザーに代わってアクションを実行できるようにするサービスにリンクされたロールにアタッチされます。詳細については、「AWS AppSync のサービスにリンクされたロール」を参照してください。

アクセス許可の詳細

このポリシーには、以下のアクセス許可が含まれています。

- X-Ray – AWS AppSync は AWS X-Ray を使用して、アプリケーション内で行われたリクエストに関するデータを収集します。詳細については、「AWS X-Ray を使用して AWS AppSync でリクエストをトレースする」を参照してください。

  このポリシーは以下のアクションを許可します。

  - サンプリングルールとその結果の取得
  - X-Ray デーモンへのトレースデータの送信

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "xray:PutTraceSegments",
                "xray:PutTelemetryRecords",
                "xray:GetSamplingTargets",
                "xray:GetSamplingRules",
                "xray:GetSamplingStatisticSummaries"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

管理ポリシーに対する AWSAWS AppSync の更新

このサービスがこれらの変更の追跡を開始してからの AWS AppSync の AWS マネージドポリシーの更新に関する詳細を表示します。このページの変更に関する自動アラートについては、 AWS AppSync ドキュメント履歴ページの RSS フィードにサブスクライブしてください。

変更	説明	日付
AWSAppSyncSchemaAuthor - 既存のポリシーに対する更新	ランタイムとコンテキストを使用したコードの評価を許可する `EvaluateCode` ポリシーアクションを追加しました。	2023 年 2 月 7 日
AWSAppSyncSchemaAuthor - 既存のポリシーに対する更新	API のリスト、取得、作成、更新、削除の各機能を許可するポリシーアクションを追加しました。 リクエストとレスポンスのリゾルバーのマッピングテンプレートロジックに対するユーザーの評価を許可す	2022 年 8 月 25 日

変更	説明	日付
	る EvaluateMappingTemplate ポリシーアクションを追加しました。 リソースのタグ付けを許可するポリシーアクションを追加しました。	
AWS AppSync が変更の追跡を開始しました	AWS AppSync は、 AWS 管理ポリシーの変更の追跡を開始しました。	2022 年 8 月 25 日

AWS AppSync 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

以下の情報は、 AWS AppSync と IAM の使用時に発生する可能性がある一般的な問題の診断と修正に役立ちます。

AWS AppSync 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と AWS Management Console 通知された場合は、管理者に連絡してサポートを依頼する必要があります。担当の管理者はお客様のユーザー名とパスワードを発行した人です。

以下のエラー例は、IAM ユーザー mateojackson がコンソールを使用して架空の *my-example-widget* リソースに関する詳細情報を表示しようとしているが、架空の appsync:*GetWidget* 許可がないという場合に発生します。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 appsync:GetWidget on resource: my-example-widget
```

この場合、Mateo は、appsync:*GetWidget* アクションを使用して *my-example-widget* リソースにアクセスできるように、管理者にポリシーの更新を依頼します。

iam:PassRole を実行する権限がない

iam:PassRole アクションを実行する権限がないという エラーが表示された場合は、ポリシーを更新して AWS AppSync 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 AWS AppSync でアクションを実行しようする場合に発生します。ただし、このアクションをサービスが実行するには、サービスロールから付与された権限が必要です。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。サインイン資格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ユーザーに付与できます 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 AWS AppSync へのアクセスを他のユーザーに許可したい

他のユーザーが AWS AppSync にアクセスできるようにするには、アクセスを必要とするユーザーまたはアプリケーションにアクセス許可を付与する必要があります。 AWS IAM Identity Center を使用してユーザーとアプリケーションを管理する場合は、アクセスレベルを定義するアクセス許可セットをユーザーまたはグループに割り当てます。アクセス許可セットは、ユーザーまたはアプリケーションに関連付けられている IAM ロールに自動的に IAM ポリシーを作成して割り当てます。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。

IAM アイデンティティセンターを使用していない場合は、アクセスを必要としているユーザー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。次に、 AWS AppSync で正しいアクセス許可を付与するポリシーをエンティティにアタッチする必要があります。アクセス許可が付与されたら、ユーザーまたはアプリケーション開発者に認証情報を提供します。これらの認証情報を使用して AWSにアクセスします。IAM ユーザー、グループ、ポリシー、アクセス許可の作成の詳細については、「IAM ユーザーガイド」の「IAM アイデンティティ」と「IAM のポリシーとアクセス許可」を参照してください。

自分の AWS アカウント以外のユーザーに my AWS AppSync 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:

- AWS AppSync がこれらの機能をサポートしているかどうかを確認するには、「」を参照してくださいIAM での AWS AppSync の仕組み。
- 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。

を使用した Logging AWS AppSync API コール AWS CloudTrail

AWS AppSync は、 AWS AppSync のユーザー AWS CloudTrail、ロール、または のサービスによって実行されたアクションを記録する AWS サービスである と統合されています。CloudTrail は AWS AppSync の API 呼び出し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 AppSync コンソールからの呼び出しと AWS AppSync 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 AppSync 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することができます。証跡を設定しない場合でも、CloudTrail コンソールの [イベント履歴] で最新のイベントを表示できます。CloudTrail で収集された情報を使用して、 AWS AppSync に対するリクエスト、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 のAWS AppSync 情報

CloudTrail は、 AWS アカウントの作成時にアカウントで有効になります。 AWS AppSync 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の他の 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 AppSync のイベントなど、 AWS アカウントのイベントの継続的な記録については、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、コンソールで証跡を作成すると、証跡はすべての AWS リージョンに適用されます。証跡は、 AWS パーティション内のすべてのリージョンからのイベントを口グに記録し、指定した Amazon S3 バケットにログファイルを配信します。さらに、CloudTrail ログで収集されたイベントデータをさらに分析して処理するように他の AWS サービスを設定できます。詳細については、次を参照してください:

- 追跡を作成するための概要

- 「CloudTrail がサポートされているサービスと統合」
- 「CloudTrail の Amazon SNS 通知の設定」
- 複数のリージョンから CloudTrail ログファイルを受け取るおよび複数のアカウントから CloudTrail ログファイルを受け取る

AWS AppSync は、 AWS AppSync API を介して行われた呼び出しのログ記録をサポートしています。現時点では、APIs への呼び出しとリゾルバーへの呼び出しは、 AWS AppSync によって CloudTrail に記録されません。

各イベントまたはログエントリには、誰がリクエストを生成したかという情報が含まれます。アイデンティティ情報は、以下を判別するのに役立ちます。

- リクエストがルートまたは AWS Identity and Access Management (IAM) ユーザー認証情報を使用して行われたかどうか。
- リクエストがロールまたはフェデレーションユーザーのテンポラリなセキュリティ認証情報を使用して行われたかどうか。
- リクエストが別の AWS サービスによって行われたかどうか。

詳細については、「CloudTrail userIdentity エレメント」を参照してください。

AWS AppSync ログファイルエントリについて

「トレイル」は、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するように設定できます。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 呼び出し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、 AWS AppSync コンソールを介して行われた`GetGraphqlApi`アクションを示す CloudTrail ログエントリを示しています。

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "ABCDEFXAMPLEPRINCIPAL:nikkiwolf",
        "arn": "arn:aws:sts::111122223333:assumed-role/admin/nikkiwolf",
        "accountId": "111122223333",
```

```
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AIDAJ45Q7YFFAREXAMPLE",
                "arn": "arn:aws:iam::111122223333:role/admin",
                "accountId": "111122223333",
                "userName": "admin"
            },
            "webIdFederationData": {},
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2021-03-12T22:41:48Z"
            }
        }
    },
    "eventTime": "2021-03-12T22:46:18Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "GetGraphqlApi",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "203.0.113.69",
    "userAgent": "aws-internal/3 aws-sdk-java/1.11.942
 Linux/4.9.230-0.1.ac.223.84.332.metal1.x86_64 OpenJDK_64-Bit_Server_VM/25.282-b08
 java/1.8.0_282 vendor/Oracle_Corporation",
    "requestParameters": {
        "apiId": "xhxt3typtfnmidkhcexampleid"
    },
    "responseElements": null,
    "requestID": "2fc43a35-a552-4b5d-be6e-12553a03dd12",
    "eventID": "b95b0ad9-8c71-4252-a2ec-5dc2fe5f8ae8",
    "readOnly": true,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "eventCategory": "Management",
    "recipientAccountId": "111122223333"
}
```

次の例は、 AWS CLIでおこなわれた `CreateApikey` アクションを示す CloudTrail ログエントリです。

```
{
    "eventVersion": "1.08",
    "userIdentity": {
```

```
        "type": "IAMUser",
        "principalId": "ABCDEFXAMPLEPRINCIPAL",
        "arn": "arn:aws:iam::111122223333:user/nikkiwolf",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "nikkiwolf"
    },
    "eventTime": "2021-03-12T22:49:10Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "CreateApiKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "203.0.113.69",
    "userAgent": "aws-cli/2.0.11 Python/3.7.4 Darwin/18.7.0 botocore/2.0.0dev15",
    "requestParameters": {
        "apiId": "xhxt3typtfnmidkhcexampleid"
    },
    "responseElements": {
        "apiKey": {
            "id": "***",
            "expires": 1616191200,
            "deletes": 1621375200
        }
    },
    "requestID": "e152190e-04ba-4d0a-ae7b-6bfc0bcea6af",
    "eventID": "ba3f39e0-9d87-41c5-abbb-2000abcb6013",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "eventCategory": "Management",
    "recipientAccountId": "111122223333"
}
```


のセキュリティのベストプラクティス AWS AppSync

保護 AWS AppSync とは、いくつかのレバーをオンにしたり、ログ記録を設定したりするだけのことではありません。以下のセクションでは、サービスの使用方法によって異なるセキュリティのベストプラクティスについて説明します。

認証方法を理解する

AWS AppSync には、GraphQL APIs に対してユーザーを認証する複数の方法が用意されています。それぞれの方法にセキュリティ、監査可能性、ユーザビリティの面でトレードオフがあります。

次の一般的な認証方法を使用できます。

- Amazon Cognito ユーザープールを使用すると、GraphQL API がユーザー属性を使用してきめ細かなアクセス制御とフィルタリングを行うことができます。
- API トークンの寿命は限られており、継続的インテグレーションシステム、外部 API との統合などの自動化システムに適しています。
- AWS Identity and Access Management (IAM) は、 で管理される内部アプリケーションに適しています AWS アカウント。
- OpenID Connect を使用すると、OpenID Connect プロトコルによってアクセスを制御およびフェデレーションできます。

AWS AppSyncでの認証および認可の詳細については、「GraphQL API を保護するための認可と認証の設定」を参照してください。

API 設定の変更がどのように伝達されるかを理解する

API 設定に変更を保存すると、 は変更の伝達 AWS AppSync を開始します。設定変更が反映されるまで、 は AWS AppSync 引き続き以前の設定のコンテンツを提供します。設定変更が伝播されると、 AWS AppSync はすぐに新しい設定に基づいてコンテンツの提供を開始します。 AWS AppSync が API の変更を伝達している間、API が以前の設定または新しい設定に基づいてコンテンツを提供しているかどうかを判断できません。

HTTP リゾルバーに TLS を使用する

HTTP リゾルバーを使用する場合は、可能な限り TLS セキュア (HTTPS) 接続を使用してください。が AWS AppSync 信頼する TLS 証明書の完全なリストについては、「」を参照してくださいHTTPS エンドポイント AWS AppSync で によって認識される認証機関 (CA)。

最小の権限を持つロールを使用する

DynamoDB リゾルバーのようなリゾルバーを使用する場合は、Amazon DynamoDB テーブルなどのリソースに対して可能な限り制限の厳しいビューを提供するロールを使用します。

IAM ポリシーのベストプラクティス

ID ベースのポリシーは、アカウント内で誰かが AWS AppSync リソースを作成、アクセス、または削除できるかどうかを決定します。これらのアクションを実行すると、 AWS アカウントに料金が発

生する可能性があります。アイデンティティベースポリシーを作成したり編集したりする際には、以下のガイドラインと推奨事項に従ってください:

- AWS 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の付与を開始するには、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することで、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、サービスアクションが などの特定の を通じて使用されている場合に 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 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 でポリシーを検証する」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、MFA をオンにしてセキュリティを強化します。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 を使用した安全な API アクセス」を参照してください。

IAM でのベストプラクティスの詳細については、「IAM ユーザーガイド」の「IAM でのセキュリティのベストプラクティス」を参照してください。

AWS AppSync リゾルバーリファレンス (JavaScript)

次のセクションには、APPSYNC_JS ランタイムと JavaScript リゾルバーリファレンスが含まれています。

- JavaScript リゾルバーの概要 - リゾルバーの AWS AppSyncでの仕組みについて詳しく説明します。
- リゾルバーコンテキストオブジェクトリファレンス - コンテキストオブジェクトの詳細とリゾルバーでの使用方法について説明します。
- リゾルバーと関数の JavaScript ランタイム機能 - サポートされているランタイム機能の詳細と、コードを簡素化するためのユーティリティの使用について説明します。
- DynamoDB 用 JavaScript リゾルバー関数リファレンス - リゾルバーが DynamoDB とやり取りする方法の詳細について説明します。
- OpenSearch 用 JavaScript リゾルバー関数リファレンス - リゾルバーのリクエストとレスポンスの構造、および OpenSearch Service とのやり取りの詳細について説明します。
- Lambda 用 JavaScript リゾルバー関数リファレンス - リゾルバーのリクエストとレスポンスの構造、および Lambda とのやり取りについて詳しく説明します。
- EventBridge データソース用 JavaScript リゾルバー関数リファレンス - リゾルバーのリクエストとレスポンスの構造、および EventBridge とのやり取りの詳細について説明します。
- None データソース用 JavaScript リゾルバー関数リファレンス - リゾルバーのリクエストとレスポンスの構造、および NONE データソースとのやり取りについて詳しく説明します。
- HTTP 用 JavaScript リゾルバー関数リファレンス - リゾルバーのリクエストとレスポンスの構造、および HTTP エンドポイントとのやり取りの詳細について説明します。
- Amazon RDS 用 JavaScript リゾルバー関数リファレンス - リゾルバーの構造と RDS とのやり取りの詳細について説明します。
- Amazon Bedrock の JavaScript リゾルバー関数リファレンス - リゾルバーの構造と Amazon Bedrock とのやり取りについて詳しく説明します。

AWS AppSync JavaScript リゾルバーの概要

AWS AppSync では、データソースに対してオペレーションを実行して GraphQL リクエストに応答できます。クエリ、ミューテーション、サブスクリプションなど、実行する GraphQL フィールドごとに、リゾルバーをアタッチする必要があります。

リゾルバーは GraphQL とデータソースの間のコネクタです。受信した GraphQL リクエストをバックエンドデータソースの手順に変換する方法と、そのデータソースからのレスポンスを GraphQL レスポンスに変換する方法を AWS AppSync に伝えます。 AWS AppSync を使用すると、JavaScript を使用してリゾルバーを記述し、 AWS AppSync (APPSYNC_JS) 環境で実行できます。

AWS AppSync では、パイプライン内の複数の AWS AppSync 関数で構成されるユニットリゾルバーまたはパイプラインリゾルバーを記述できます。

サポートされているランタイム機能

AWS AppSync JavaScript ランタイムは、JavaScript ライブラリ、ユーティリティ、機能のサブセットを提供します。APPSYNC_JS ランタイムでサポートされている特徴と機能の完全なリストについては、「リゾルバーと関数用の JavaScript ランタイム機能」を参照してください。

ユニットリゾルバー

ユニットリゾルバーは、データソースに対して実行されるリクエストハンドラーとレスポンスハンドラーを定義するコードで構成されています。リクエストハンドラーはコンテキストオブジェクトを引数として受け取り、データソースの呼び出しに使用されたリクエストペイロードを返します。レスポンスハンドラーは、実行されたリクエストの結果を含むペイロードをデータソースから受け取ります。レスポンスハンドラーは、ペイロードを GraphQL レスポンスに変換して GraphQL フィールドを解決します。以下の例では、リゾルバーは DynamoDB データソースから項目を取得します。

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  return ddb.get({ key: { id: ctx.args.id } });
}

export const response = (ctx) => ctx.result;
```

JavaScript パイプラインリゾルバーの仕組み

パイプラインリゾルバーは、リクエストとレスポンスのハンドラーと関数のリストを定義するコードで構成されています。各関数には、データソースに対して実行されるリクエストおよびレスポンスハンドラーがあります。パイプラインリゾルバーは実行をリスト内の関数に委任するため、いずれのデータソースにもリンクされていません。ユニットリゾルバーと関数は、データソースに対してオペレーションを実行するプリミティブです。

パイプラインリゾルバーリクエストハンドラー

パイプラインリゾルバーのリクエストハンドラー (before step) では、定義した関数を実行する前に準備ロジックを実行できます。

関数のリスト

パイプラインリゾルバーが順番に実行する関数のリスト。パイプラインリゾルバーのリクエストハンドラーの評価結果は、最初の関数で `ctx.prev.result` として使用可能になります。各関数の評価結果は、以下の関数で `ctx.prev.result` として使用可能です。

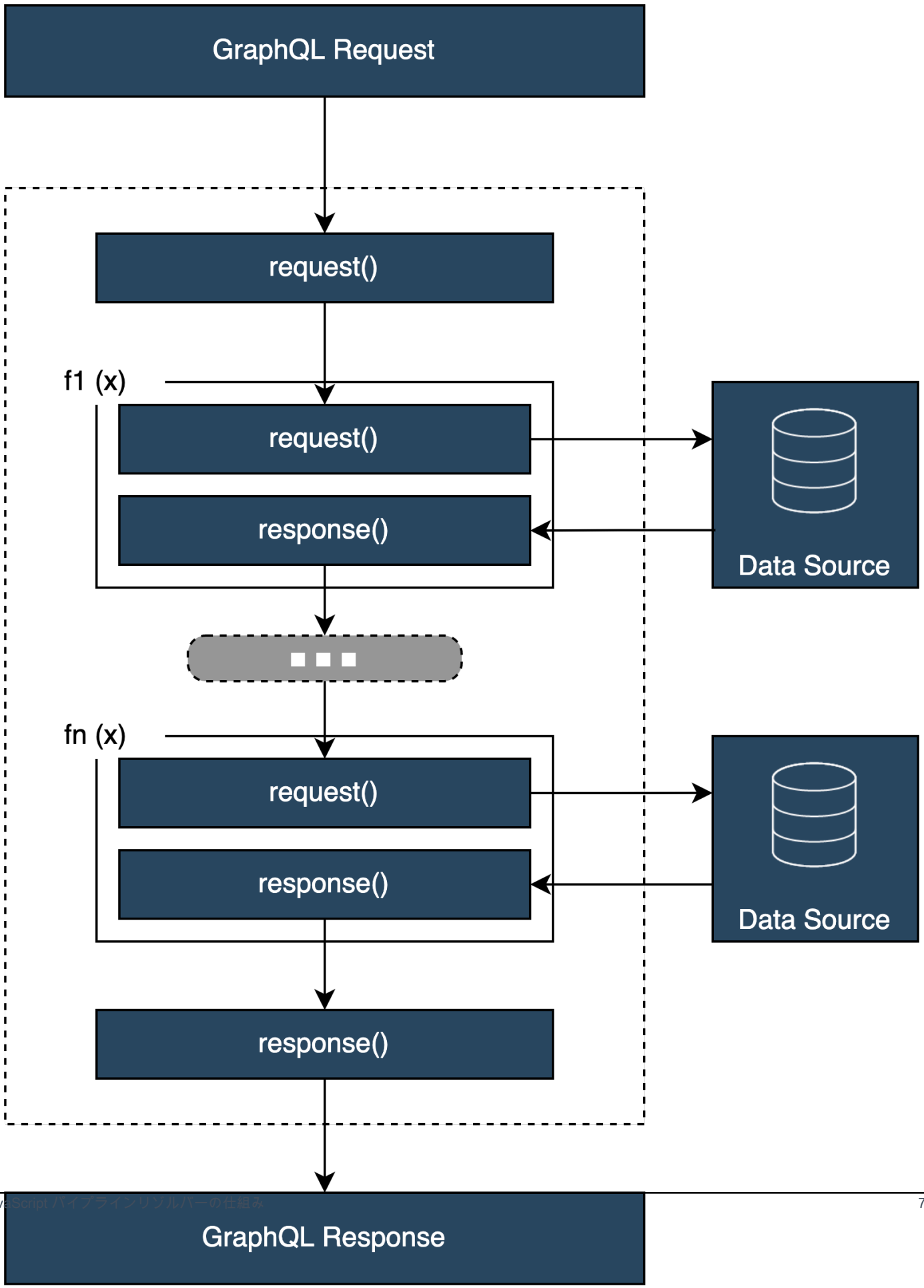
パイプラインリゾルバーレスポンスハンドラー

パイプラインリゾルバーのレスポンスハンドラーでは、最後の関数の出力から、想定される GraphQL フィールドタイプへの、一部の最終的なロジックを実行できます。関数のリストの最後にある関数の出力は、パイプラインリゾルバーレスポンスハンドラーで `ctx.prev.result` または `ctx.result` として使用可能です。

実行フロー

2 つの関数で構成されるパイプラインリゾルバーがあるとします。以下のリストでは、リゾルバーが呼び出されたときの実行フローを表しています。

1. パイプラインリゾルバーリクエストハンドラー
2. 関数 1: 関数リクエストハンドラー
3. 関数 1: データソースの呼び出し
4. 関数 1: 関数レスポンスハンドラー
5. 関数 2: 関数リクエストハンドラー
6. 関数 2: データソースの呼び出し
7. 関数 2: 関数レスポンスハンドラー
8. パイプラインリゾルバーレスポンスハンドラー


GraphQL Request
request()
f1 (x)
request()
response()
Data Source
fn (x)
request()
response()
Data Source
response()
GraphQL Response

便利な APPSYNC_JS ランタイムビルトインユーティリティ

パイプラインリゾルバーでの作業には、以下のユーティリティが役立ちます。

ctx.stash

stash は、各リゾルバー、関数リクエスト、レスポンスハンドラー内で使用できるオブジェクトです。同じ stash インスタンスは 1 つのリゾルバーの実行を通して存続します。つまり、stash を使用して、リクエストとレスポンスのハンドラー間、およびパイプラインリゾルバーの関数間で、任意のデータを渡すことができます。stash は通常の JavaScript オブジェクトと同じようにテストできます。

ctx.prev.result

`ctx.prev.result` は、パイプラインで実行された前のオペレーションの結果を表します。前のオペレーションがパイプラインリゾルバーリクエストハンドラーであった場合、`ctx.prev.result` はチェーン内の最初の関数に使用可能になります。前のオペレーションが最初の関数であった場合、`ctx.prev.result` は最初の関数の出力を表し、パイプライン内の 2 番目の関数に使用可能になります。前のオペレーションが最後の関数であった場合、`ctx.prev.result` は最後の関数の出力を表し、パイプラインリゾルバーレスポンスハンドラーに使用可能になります。

util.error

`util.error` ユーティリティはフィールドエラーをスローするのに便利です。関数リクエストまたはレスポンスハンドラー内で `util.error` を使用すると、フィールドエラーがすぐにスローされ、それ以降の関数が実行されなくなります。詳細やその他の `util.error` シグネチャについては、「リゾルバーと関数の JavaScript ランタイム機能」を参照してください。

util.appendError

`util.appendError` は `util.error()` と似ていますが、ハンドラーの評価を中断しないという大きな違いがあります。代わりに、フィールドにエラーがあったことを通知します。ただし、ハンドラーを評価し、データを引き続き返すことも可能です。関数内で `util.appendError` を使用しても、パイプラインの実行フローは中断されません。詳細やその他の `util.error` シグネチャについては、「リゾルバーと関数の JavaScript ランタイム機能」を参照してください。

runtime.earlyReturn

`runtime.earlyReturn` 関数を使うと、どのリクエスト関数からでも途中で戻ることができます。リゾルバーのリクエストハンドラー内で `runtime.earlyReturn` を使用すると、リゾルバーから返

されます。 AWS AppSync 関数リクエストハンドラーから呼び出すと、関数から戻り、パイプライン内の次の関数またはリゾルバー応答ハンドラーのどちらかまで実行が続行されます。

パイプラインリゾルバーの記述

パイプラインリゾルバーには、パイプライン内の関数の実行を囲むリクエストハンドラーとレスポンスハンドラーもあります。リクエストハンドラーは最初の関数のリクエストの前に実行され、レスポンスハンドラーは最後の関数の応答の後に実行されます。リゾルバーリクエストハンドラーは、パイプライン内の関数が使用するデータを設定できます。リゾルバーレスポンスハンドラーは GraphQL フィールド出力タイプにマッピングするデータを返す役割を果たします。以下の例では、リゾルバーリクエストハンドラーが `allowedGroups` を定義しています。返されるデータはこれらのグループのいずれかに属している必要があります。この値は、リゾルバーの関数がデータをリクエストする際に使用できます。リゾルバーのレスポンスハンドラーは最終チェックを行い、結果をフィルタリングして、許可されたグループに属するアイテムだけが返されるようにします。

```
import { util } from '@aws-appsync/utils';

/**
 * Called before the request function of the first AppSync function in the pipeline.
 *  @param ctx the context object holds contextual information about the function
 invocation.
 */
export function request(ctx) {
  ctx.stash.allowedGroups = ['admin'];
  ctx.stash.startedAt = util.time.nowISO8601();
  return {};
}
/**
 * Called after the response function of the last AppSync function in the pipeline.
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function response(ctx) {
  const result = [];
  for (const item of ctx.prev.result) {
    if (ctx.stash.allowedGroups.indexOf(item.group) > -1) result.push(item);
  }
  return result;
}
```

AWS AppSync 関数の記述

AWS AppSync 関数を使用すると、スキーマ内の複数のリゾルバー間で再利用できる一般的なロジックを記述できます。例えば、Amazon DynamoDB データソースからの項目のクエリQUERY_ITEMSを担当する という 1 つの AWS AppSync 関数を持つことができます。項目をクエリするリゾルバーについては、リゾルバーのパイプラインに関数を追加し、使用するクエリインデックスを指定するだけです。ロジックを再実装する必要はありません。

補足トピック

トピック

- Amazon DynamoDB を使用したパイプラインリゾルバーの例
- APPSYNC_JS ランタイムのユーティリティの設定
- APPSYNC_JS ランタイムのバンドル、TypeScript、ソースマップ
- リゾルバーと関数ハンドラーのテスト
- VTL から JavaScript への移行
- データソースへの直接アクセスと Lambda データソース経由のプロキシのどちらかを選択する

Amazon DynamoDB を使用したパイプラインリゾルバーの例

次の GraphQL クエリを使用して、Amazon DynamoDB データソースから `Post` 型を返す `getPost(id:ID!)` という名前のフィールドにパイプラインリゾルバーをアタッチするとします。

```
getPost(id:1){
    id
    title
    content
}
```

まず、以下のコードで簡単なリゾルバーを `Query.getPost` にアタッチします。これは単純なリゾルバーコードの例です。リクエストハンドラーにはロジックは定義されておらず、レスポンスハンドラーは最後の関数の結果を返すだけです。

```
/**
 * Invoked **before** the request handler of the first AppSync function in the
 pipeline.
 * The resolver `request` handler allows to perform some preparation logic
```

```
 * before executing the defined functions in your pipeline.
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function request(ctx) {
  return {}
}

/**
 * Invoked **after** the response handler of the last AppSync function in the pipeline.
 * The resolver `response` handler allows to perform some final evaluation logic
 * from the output of the last function to the expected GraphQL field type.
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function response(ctx) {
  return ctx.prev.result
}
```

次に、データソースから投稿項目を取得する関数 GET_ITEM を定義します。

```
import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'

/**
 * Request a single item from the attached DynamoDB table datasource
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function request(ctx) {
 const { id } = ctx.args
 return ddb.get({ key: { id } })
}

/**
 * Returns the result
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function response(ctx) {
 const { error, result } = ctx
 if (error) {
  return util.appendError(error.message, error.type, result)
```

```
 }
 return ctx.result
}
```

リクエスト中にエラーが発生した場合、関数のレスポンスハンドラーは、呼び出し元のクライアントに返されるエラーを GraphQL レスポンスに追加します。GET_ITEM 関数をリゾルバー関数リストに追加します。クエリを実行すると、GET_ITEM関数のリクエストハンドラーは DynamoDB モジュールが提供する utils AWS AppSyncを使用して、 をキーidとして使用してDynamoDBGetItemリクエストを作成します。 は適切なGetItemオペレーションddb.get({ key: { id } })を生成します。

```
{
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "1" }
    }
}
```

AWS AppSync は リクエストを使用して Amazon DynamoDB からデータを取得します。データが返されると、GET_ITEM 関数のレスポンスハンドラーによって処理されます。レスポンスハンドラーはエラーをチェックして結果を返します。

```
{
  "result" : {
    "id": 1,
    "title": "hello world",
    "content": "<long story>"
  }
}
```

最後に、リゾルバーのレスポンスハンドラーは結果を直接返します。

エラーの使用

リクエスト中に関数でエラーが発生した場合、そのエラーは ctx.error の関数レスポンスハンドラーで利用できるようになります。util.appendError ユーティリティを使用して GraphQL レスポンスにエラーを追加できます。stash を使用すると、パイプライン内の他の関数がエラーを利用できるようになります。次の例を参照してください。

```
/**
```

```
 * Returns the result
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    if (!ctx.stash.errors) ctx.stash.errors = []
    ctx.stash.errors.push(ctx.error)
    return util.appendError(error.message, error.type, result);
  }
  return ctx.result;
}
```

APPSYNC_JS ランタイムのユーティリティの設定

AWS AppSync には、 APPSYNC_JSランタイムを使用したリゾルバーの開発に役立つ 2 つのライブラリが用意されています。

- `@aws-appsync/eslint-plugin` - 開発中に問題を迅速に発見して修正します。
- `@aws-appsync/utils` - コードエディターで型検証とオートコンプリートを行います。

eslint プラグインの設定

ESLint は、コードを静的に分析して問題をすばやく見つけるツールです。ESLint は継続的インテグレーションパイプラインの一部として実行できます。`@aws-appsync/eslint-plugin`は、APPSYNC_JS ランタイムを利用する際にコード内の無効な構文を検出する ESLint プラグインです。このプラグインを使用すると、変更をクラウドにプッシュしなくても、開発中にコードに関するフィードバックを迅速に得ることができます。

`@aws-appsync/eslint-plugin` には、開発中に使用できる 2 つのルールセットが用意されています。

"plugin:@aws-appsync/base" は、プロジェクトで活用できる基本ルールセットを設定します。

ルール	説明
非同期	非同期のプロセスと Promise はサポートされていません。

ルール	説明
no-await	非同期のプロセスと Promise はサポートされていません。
no-classes	ルールはサポートされていません。
no-for	`for` はサポートされていない (サポートされている `for-in` および `for-of` は除く)
no-continue	`continue` はサポートされていません。
no-generators	ジェネレータはサポートされていません。
no-yield	`yield` はサポートされていません。
no-labels	ラベルはサポートされていません。
no-this	`this` キーワードはサポートされていません。
no-try	try/catch 構造はサポートされていません。
no-while	while ループはサポートされていません。
no-disallowed-unary-operators	`++`、`--` および ~ 単項演算子は使用できません。
no-disallowed-binary-operators	`instanceof` 演算子は使用できません。
no-promise	非同期のプロセスと Promise はサポートされていません。

"plugin:@aws-appsync/recommended" にはいくつかの追加のルールがありますが、プロジェクトに TypeScript 構成を追加する必要もあります。

ルール	説明
no-recursion	再帰的な関数呼び出しは許可されません。

ルール	説明
no-disallowed-methods	使用できないメソッドもあります。サポートされている組み込み関数の全セットについては、リファレンスを参照してください。
no-function-passing	関数を関数の引数として関数に渡すことはできません。
no-function-reassign	関数は再割り当てできません。
no-function-return	関数を関数の戻り値にすることはできません。

プラグインをプロジェクトに追加するには、「ESLint 入門」のインストールと使用手順に従ってください。次に、プロジェクトパッケージマネージャー (npm、yarn、pnpm など) を使用してプロジェクトにプラグインをインストールします。

```
$ npm install @aws-appsync/eslint-plugin
```

`.eslintrc.{js,yml,json}` ファイルで "plugin:@aws-appsync/base"または "plugin:@aws-appsync/recommended"を `extends` プロパティに追加します。以下のスニペットは JavaScript の基本的なサンプル `.eslintrc` 設定です。

```
{
  "extends": ["plugin:@aws-appsync/base"]
}
```

"plugin:@aws-appsync/recommended" ルールセットを使用するには、必要な依存関係をインストールします。

```
$ npm install -D @typescript-eslint/parser
```

`.eslintrc.js` ファイルを作成するには

```
{
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaVersion": 2018,
```

```
    "project": "./tsconfig.json"
  },
  "extends": ["plugin:@aws-appsync/recommended"]
}
```

APPSYNC_JS ランタイムのバンドル、TypeScript、ソースマップ

TypeScript は、型の安全性と早期エラー検出を提供することで AWS AppSync 開発を強化します。TypeScript コードをローカルで記述し、APPSYNC_JS ランタイムで使用する前に JavaScript にトランスパイルできます。このプロセスは、TypeScript のインストールと APPSYNC_JS 環境の tsconfig.json の設定から始まります。その後、esbuild などのバンドルツールを使用して、コードをコンパイルしてバンドルできます。Amplify CLI は GraphQL スキーマからタイプを生成し、リゾルバーコードでこれらのタイプを使用できます。

リゾルバーと関数コードでは、APPSYNC_JS 要件を満たす限り、カスタムライブラリと外部ライブラリの両方を活用できます。バンドルツールは、コードを 1 つのファイルに結合して使用します AWS AppSync。ソースマップを含めると、デバッグに役立ちます。

ライブラリの活用とコードのバンドル

リゾルバーと関数コードでは、APPSYNC_JS 要件を満たす限り、カスタムライブラリと外部ライブラリの両方を活用できます。これにより、アプリケーション内の既存のコードを再利用できます。複数のファイルで定義されたライブラリを使用するには、esbuild などのバンドルツールを使用してコードを 1 つのファイルに結合し、リ AWS AppSync ゾルバーまたは関数に保存する必要があります。

コードをバンドルするときは、次の点に留意してください。

- APPSYNC_JS は、ECMAScript モジュール (ESM) のみをサポートします。
- @aws-appsync/* モジュールは APPSYNC_JS に統合されており、バンドルすべきではありません。
- APPSYNC_JS ランタイム環境は、コードがブラウザ環境では実行されないという点で NodeJS に似ています。
- オプションでソースマップを含めることができます。ただし、ソースコンテンツを含めないでください。

  ソースマップの詳細については、「ソースマップの使用」を参照してください。

例えば、src/appsync/getPost.resolver.js にあるリゾルバーコードをバンドルするには、次の esbuild CLI コマンドを使用することができます。

```
$ esbuild --bundle \
--sourcemap=inline \
--sources-content=false \
--target=esnext \
--platform=node \
--format=esm \
--external:@aws-appsync/utils \
--outdir=out/appsync \
 src/appsync/getPost.resolver.js
```

コードのビルドと TypeScript での作業

TypeScript は Microsoft が開発したプログラミング言語で、TypeScript タイピングシステムとともに JavaScript のすべての機能を備えています。TypeScript を使用すると、タイプセーフなコードを記述し、ビルド時にコードを AWS AppSyncに保存する前にエラーやバグを検出できます。@aws-appsync/utils パッケージは完全に型指定されています。

APPSYNC_JS ランタイムは TypeScript を直接サポートしていません。コードを AWS AppSyncに保存する前に、まず TypeScript コードを APPSYNC_JS ランタイムがサポートする JavaScript コードにトランスパイルする必要があります。TypeScript を使用してローカル統合開発環境 (IDE) にコードを記述できますが、 AWS AppSync コンソールで TypeScript コードを作成することはできません。

開始する前に、プロジェクトに TypeScript がインストールされていることを確認してください。次に、TSConfig を使用して APPSYNC_JS ランタイムと連携するように TypeScript のトランスコンパイル設定を構成します。使用できる基本 tsconfig.json ファイルの例を次に示します。

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "esnext",
    "module": "esnext",
   "noEmit": true,
   "moduleResolution": "node",
  }
}
```

その後、esbuild などのバンドルツールを使用して、コードをコンパイルしてバンドルできます。たとえば、 にある AWS AppSync コードを持つプロジェクトの場合src/appsync、次のコマンドを使用してコードをコンパイルしてバンドルできます。

```
$ esbuild --bundle \
--sourcemap=inline \
--sources-content=false \
--target=esnext \
--platform=node \
--format=esm \
--external:@aws-appsync/utils \
--outdir=out/appsync \
 src/appsync/**/*.ts
```

Amplify Codegen を使用する

Amplify CLI を使用してスキーマのタイプを生成できます。schema.graphql ファイルが置かれているディレクトリから以下のコマンドを実行し、プロンプトを確認して codegen を設定します。

```
$  npx @aws-amplify/cli codegen add
```

特定の状況 (スキーマの更新時など) で codegen を再生成するには、以下のコマンドを実行します。

```
$ npx @aws-amplify/cli codegen
```

その後、生成された型をリゾルバーコードで使用できます。例として、次のスキーマがあるとします。

```
type Todo {
  id: ID!
  title: String!
  description: String
}

type Mutation {
  createTodo(title: String!, description: String): Todo
}

type Query {
  listTodos: Todo
```

```
}
```

生成された型は、次の例の AWS AppSync 関数で使用できます。

```
import { Context, util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'
import { CreateTodoMutationVariables, Todo } from './API' // codegen

export function request(ctx: Context<CreateTodoMutationVariables>) {
 ctx.args.description = ctx.args.description ?? 'created on ' + util.time.nowISO8601()
 return ddb.put<Todo>({ key: { id: util.autoId() }, item: ctx.args })
}

export function response(ctx) {
 return ctx.result as Todo
}
```

TypeScript でのジェネリックの使用

ジェネリックスは用意されているいくつかの型で使用できます。例えば、以下のスニペットは Todo タイプです。

```
export type Todo = {
  __typename: "Todo",
  id: string,
  title: string,
  description?: string | null,
};
```

Todo を利用するサブスクリプション用のリゾルバーを書くことができます。お使いの IDE では、型定義とオートコンプリートのヒントを参考にして、toSubscriptionFilter 変換ユーティリティを適切に使用してください。

```
import { util, Context, extensions } from '@aws-appsync/utils'
import { Todo } from './API'

export function request(ctx: Context) {
  return {}
}

export function response(ctx: Context) {
```

```
  const filter = util.transform.toSubscriptionFilter<Todo>({
    title: { beginsWith: 'hello' },
    description: { contains: 'created' },
  })
  extensions.setSubscriptionFilter(filter)
  return null
}
```

バンドルのリンティング

esbuild-plugin-eslint プラグインをインポートすることで、バンドルを自動的にリントできます。その後、eslint 機能を有効にする plugins 値を指定することで有効化できます。以下は、esbuild JavaScript API を build.mjs というファイルで使用しているスニペットです。

```
/* eslint-disable */
import { build } from 'esbuild'
import eslint from 'esbuild-plugin-eslint'
import glob from 'glob'
const files = await glob('src/**/*.ts')

await build({
  format: 'esm',
  target: 'esnext',
  platform: 'node',
  external: ['@aws-appsync/utils'],
  outdir: 'dist/',
  entryPoints: files,
  bundle: true,
  plugins: [eslint({ useEslintrc: true })],
})
```

ソースマップの使用

JavaScript コードでインラインソースマップ (sourcemap) を提供できます。ソースマップは、JavaScript または TypeScript コードをバンドルしていて、入力ソースファイルへの参照をログやランタイム JavaScript エラーメッセージに表示したい場合に便利です。

sourcemap はコードの最後に表示する必要があります。以下の形式の 1 行のコメントで定義されます。

```
//# sourceMappingURL=data:application/json; base64,<base64 encoded string>
```

例を示します。

```
//# sourceMappingURL=data:application/
json; base64,ewogICJ2ZXJzaW9uIjogMywKICAic291cmNlcyI6IFsibGliLmpzIiwgImNvZGUuanMiXSwKICAibWFwcGl
```

ソースマップは esbuild で作成できます。以下の例は、コードをビルドしてバンドルするときに esbuild JavaScript API を使用してインラインソースマップを組み込む方法を示しています。

```
/* eslint-disable */
import { build } from 'esbuild'
import eslint from 'esbuild-plugin-eslint'
import glob from 'glob'
const files = await glob('src/**/*.ts')

await build({
  sourcemap: 'inline',
  sourcesContent: false,

  format: 'esm',
  target: 'esnext',
  platform: 'node',
  external: ['@aws-appsync/utils'],
  outdir: 'dist/',
  entryPoints: files,
  bundle: true,
  plugins: [eslint({ useEslintrc: true })],
})
```

特に、sourcemap および sourcesContent オプションでは、各ビルドの最後にソースマップをインラインで追加し、ソースコンテンツは含めないように指定しています。慣例として、ソースコンテンツは sourcemap には含めないことをお勧めします。これを無効にするには、sources-content を false に設定します。

ソースマップの仕組みを説明するために、リゾルバーコードがヘルパーライブラリのヘルパー関数を参照する次の例を確認してください。このコードには、リゾルバーコードとヘルパーライブラリーのログステートメントが含まれています。

./src/default.resolver.ts (自身のリゾルバー)

```
import { Context } from '@aws-appsync/utils'
import { hello, logit } from './helper'
```

```
export function request(ctx: Context) {
  console.log('start >')
  logit('hello world', 42, true)
  console.log('< end')
  return 'test'
}

export function response(ctx: Context): boolean {
  hello()
  return ctx.prev.result
}
```

.src/helper.ts (ヘルパーファイル)

```
export const logit = (...rest: any[]) => {
  // a special logger
  console.log('[logger]', ...rest.map((r) => `<${r}>`))
}

export const hello = () => {
  // This just returns a simple sentence, but it could do more.
  console.log('i just say hello..')
}
```

リゾルバーファイルをビルドしてバンドルすると、リゾルバーコードにはインラインソースマップが含まれます。リゾルバーを実行すると、CloudWatch ログに次のエントリが表示されます。

CloudWatch ログのエントリを見ると、2 つのファイルの機能がバンドルされ、同時に実行されていることがわかります。各ファイルの元のファイル名もログにはっきりと反映されています。

でのリゾルバーと関数ハンドラーのテスト AWS AppSync

`EvaluateCode` API コマンドを使用すると、コードをリゾルバーまたは関数に保存する前に、モックデータを使用してリゾルバーと関数ハンドラーをリモートでテストできます。コマンドを使い始め

るには、ポリシーに appsync:evaluatecode アクセス許可を追加していることを確認してください。例:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateCode",
            "Resource": "arn:aws:appsync:<region>:<account>:*"
        }
    ]
}
```

AWS CLI または AWS SDK を使用してコマンドを活用できます。例えば、CLI を使用してコードをテストするには、ファイルを指定し、コンテキストを指定し、評価するハンドラーを指定するだけです。

```
aws appsync evaluate-code \
  --code file://code.js \
  --function request \
  --context file://context.json \
  --runtime name=APPSYNC_JS,runtimeVersion=1.0.0
```

レスポンスには、ハンドラーから返されたペイロードを含む evaluationResult が含まれます。また、評価中にハンドラーによって生成されたログのリストを保持する logs オブジェクトも含まれています。これにより、コード実行のデバッグが容易になり、評価に関する情報を確認してトラブルシューティングに役立てることができます。例:

```
{
    "evaluationResult": "{\"operation\":\"PutItem\",\"key\":{\"id\":{\"S\":\"record-id
\"}},\"attributeValues\":{\"owner\":{\"S\":\"John doe\"},\"expectedVersion\":{\"N\":2},
\"authorId\":{\"S\":\"Sammy Davis\"}}}",
    "logs": [
        "INFO - code.js:5:3: \"current id\" \"record-id\"",
        "INFO - code.js:9:3: \"request evaluated\""
    ]
```

```
}
```

評価結果は JSON として解析でき、以下のようになります。

```
{
  "operation": "PutItem",
  "key": {
    "id": {
      "S": "record-id"
    }
  },
  "attributeValues": {
    "owner": {
      "S": "John doe"
    },
    "expectedVersion": {
      "N": 2
    },
    "authorId": {
      "S": "Sammy Davis"
    }
  }
}
```

SDK を使用すると、テストスイートのテストを簡単に組み込んで、コードの動作を検証できます。この例では Jest テストフレームワークを使用していますが、どのテストスイートでも機能します。次のスニペットは、仮説検証の実行を示しています。評価レスポンスは有効な JSON であることを想定しているので、JSON.parseを使用して文字列レスポンスから JSON を取得することに注意してください。

```
const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })
const runtime = {name:'APPSYNC_JS',runtimeVersion:'1.0.0')

test('request correctly calls DynamoDB', async () => {
  const code = fs.readFileSync('./code.js', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateCode({ code, context, runtime, function:
 'request' }).promise()
```

```
  const result = JSON.parse(response.evaluationResult)

  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})
```

これにより、次のような結果が得られます。

```
Ran all test suites.
> jest

PASS ./index.test.js
# request correctly calls DynamoDB (543 ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 totalTime: 1.511 s, estimated 2 s
```

での VTL から JavaScript への移行 AWS AppSync

AWS AppSync を使用すると、VTL または JavaScript を使用してリゾルバーと関数のビジネスロジックを記述できます。どちらの言語でも、データソースの操作方法を AWS AppSync サービスに指示するロジックを記述します。VTL では、有効な JSON エンコードされた文字列に評価される必要があるマッピングテンプレートを作成します。JavaScript では、オブジェクトを返すリクエストハンドラーとレスポンスハンドラーを記述します。JSON エンコードされた文字列を返すことはありません。

例えば、次の VTL マッピングテンプレートを使用して Amazon DynamoDB アイテムを取得します。

```
{
    "operation": "GetItem",
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
    }
}
```

このユーティリティ `$util.dynamodb.toDynamoDBJson` は、JSON エンコードされた文字列を返します。`$ctx.args.id` を <id> に設定すると、テンプレートは有効な JSON エンコードされた文字列と評価されます。

```
{
```

```
    "operation": "GetItem",
    "key": {
        "id": {"S": "<id>"},
    }
}
```

JavaScript を使用する場合、コード内に JSON でエンコードされた生の文字列を出力する必要はなく、toDynamoDBJsonのようなユーティリティを使用する必要もありません。上記のマッピングテンプレートと同等の例を以下に示します。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: {id: util.dynamodb.toDynamoDB(ctx.args.id)}
  };
}
```

別の方法としては、util.dynamodb.toMapValues を使用する方法があります。これは、値のオブジェクトを処理する場合に推奨される方法です。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}
```

これは以下のように評価されます。

```
{
  "operation": "GetItem",
  "key": {
    "id": {
      "S": "<id>"
    }
  }
}
```

> **Note**
>
> DynamoDB モジュールを DynamoDB データソースで使用することをお勧めします。
>
> ```
> import * as ddb from '@aws-appsync/utils/dynamodb'
>
> export function request(ctx) {
>  ddb.get({ key: { id: ctx.args.id } })
> }
> ```

別の例として、次のマッピングテンプレートを使用して Amazon DynamoDB データソースに項目を配置します。

```
{
    "operation" : "PutItem",
    "key" : {
        "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

このマッピングテンプレート文字列を評価すると、有効な JSON でエンコードされた文字列が生成される必要があります。JavaScript を使用する場合、コードはリクエストオブジェクトを直接返します。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { id = util.autoId(), ...values } = ctx.args;
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}
```

以下として評価されます。

```
{
  "operation": "PutItem",
```

```
  "key": {
    "id": { "S": "2bff3f05-ff8c-4ed8-92b4-767e29fc4e63" }
  },
  "attributeValues": {
    "firstname": { "S": "Shaggy" },
    "age": { "N": 4 }
  }
}
```

> **Note**
>
> DynamoDB モジュールを DynamoDB データソースで使用することをお勧めします。
>
> ```
> import { util } from '@aws-appsync/utils'
> import * as ddb from '@aws-appsync/utils/dynamodb'
>
> export function request(ctx) {
>  const { id = util.autoId(), ...item } = ctx.args
>  return ddb.put({ key: { id }, item })
> }
> ```

データソースへの直接アクセスと Lambda データソース経由のプロキシのどちらかを選択する

AWS AppSync と APPSYNC_JSランタイムを使用すると、 AWS AppSync 関数を使用してデータソースにアクセスすることで、カスタムビジネスロジックを実装する独自のコードを記述できます。これにより、追加の計算 AWS サービスやインフラストラクチャをデプロイすることなく、Amazon DynamoDB、Aurora Serverless、OpenSearch Service、HTTP APIs、その他の サービスなどのデータソースと直接やり取りすることが容易になります。 AWS AppSync では、Lambda データソースを設定することで、 AWS Lambda 関数の操作も簡単です。Lambda データソースを使用すると、AWS Lambdaのフルセット機能を使用して複雑なビジネスロジックを実行し、GraphQL リクエストを解決できます。ほとんどの場合、ターゲットデータソースに直接接続された AWS AppSync 関数は、必要なすべての機能を提供します。APPSYNC_JS ランタイムでサポートされていない複雑なビジネスロジックを実装する必要がある状況では、Lambda データソースをプロキシとして使用してターゲットデータソースとやり取りできます。

	データソースの直接統合	プロキシとしての Lambda データソース
ユースケース	AWS AppSync functions interact directly with API data sources.	AWS AppSync functions call Lambdas that interact with API data sources.
Runtime	*APPSYNC_JS* (JavaScript)	サポートされる Lambda ランタイム
Maximum size of code	AWS AppSync 関数あたり 32,000 文字	50 MB (zip 圧縮済み、直接アップロード)
External modules	制限あり-APPSYNC_JS がサポートしている機能のみ	はい
Call any AWS service	はい - AWS AppSync HTTP データソースの使用	はい - AWS SDK の使用
Access to the request header	はい	はい
Network access	なし	あり
File system access	なし	はい
Logging and metrics	あり	あり
Build and test entirely within AppSync	はい	いいえ
Cold start	いいえ	No - プロビジョニング済み同時実行
Auto-scaling	はい - AWS AppSync によって透過的に	Yes - Lambda で設定されているとおり
Pricing	追加料金なし	Lambda の使用に対して課金

ターゲットデータソースと直接統合するAWS AppSync 関数は、次のようなユースケースに最適です。

- Amazon DynamoDB、Aurora サーバーレス、OpenSearch サービスとのやり取り
- HTTP API とのやりとりと受信ヘッダーの受け渡し
- HTTP データソースを使用して AWS サービスとやり取りする (指定されたデータソースロールを使用してリクエスト AWS AppSync に自動的に署名する)
- データソースにアクセスする前のアクセス制御の実装
- リクエストに応答する前に、取得したデータをフィルター処理する。
- リゾルバーパイプラインでの AWS AppSync 関数のシーケンシャル実行によるシンプルなオーケストレーションの実装
- クエリとミューテーションにおけるキャッシュ接続とサブスクリプション接続の制御。

Lambda データソースをプロキシとして使用するAWS AppSync 関数は、次のようなユースケースに最適です。

- JavaScript または Velocity TL (VTL) 以外の言語を使用する
- CPU またはメモリの調整と制御によるパフォーマンスの最適化
- サードパーティ製ライブラリのインポートまたは APPSYNC_JS でサポートされていない機能の使用が必要
- 複数のネットワークリクエストを行ったり、クエリを実行するためにファイルシステムにアクセスしたりする
- バッチ設定を使用してリクエストをバッチ処理する。

AWS AppSync JavaScript リゾルバーコンテキストオブジェクトリファレンス

AWS AppSync は、リクエストハンドラーとレスポンスハンドラーを操作するための変数と関数のセットを定義します。これにより、GraphQL を使用してデータの論理的オペレーションが容易になります。このドキュメントでは、それらの関数について説明し、例を示します。

`context` へのアクセス

リクエスト&レスポンスハンドラーの `context` 引数は、リゾルバー呼び出しのコンテキスト情報をすべて保持するオブジェクトです。この変数の構造は次のとおりです。

```
type Context = {
```

```
  arguments: any;
  args: any;
  identity: Identity;
  source: any;
  error?: {
    message: string;
    type: string;
  };
  stash: any;
  result: any;
  prev: any;
  request: Request;
  info: Info;
};
```

> **Note**
>
> context オブジェクトは ctx と呼ばれることがよくあります。

context オブジェクトの各フィールドは次のように定義されます。

context フィールド

arguments

このフィールドのすべての GraphQL 引数を含むマップ。

identity

呼び出し元に関する情報を含むオブジェクト。このフィールドの構造の詳細については、「ID」を参照してください。

source

親フィールドの解像度を含むマップ。

stash

stash は、リゾルバーと関数ハンドラー内部で使用可能になるオブジェクトです。リゾルバーを 1 回実行すると、同じ stash オブジェクトが存続します。つまり、stash を使用して、リクエストとレスポンスのハンドラー間、およびパイプラインリゾルバーの関数間で、任意のデータを渡すことができます。

> **Note**
>
> stash 全体を削除または置換することはできませんが、スタッシュのプロパティを追加、更新、削除、および読み取ることはできます。

以下のコード例のいずれかを変更することで、stash にアイテムを追加できます。

```
//Example 1
ctx.stash.newItem = { key: "something" }

//Example 2
Object.assign(ctx.stash, {key1: value1, key2: value})
```

以下のコードを変更することで、stashからアイテムを削除できます。

```
delete ctx.stash.key
```

`result`

このリゾルバーの結果用のコンテナ。このフィールドは、レスポンスハンドラーのみが使用できます。

例えば、次のクエリの `author` フィールドを解決する場合:

```
query {
    getPost(id: 1234) {
        postId
        title
        content
        author {
            id
            name
        }
    }
}
```

そうすれば、レスポンスハンドラーが評価されるときに `context` 変数全体が使用可能になります。

```
{
  "arguments" : {
    id: "1234"
  },
  "source": {},
  "result" : {
      "postId": "1234",
      "title": "Some title",
      "content": "Some content",
      "author": {
        "id": "5678",
        "name": "Author Name"
      }
  },
  "identity" : {
      "sourceIp" : ["x.x.x.x"],
      "userArn" : "arn:aws:iam::123456789012:user/appsync",
      "accountId" : "666666666666",
      "user" : "AIDAAAAAAAAAAAAAAAAA"
  }
}
```

prev.result

パイプラインリゾルバーで実行された前回のオペレーションの結果を表します。

前のオペレーションがパイプラインリゾルバーのリクエストハンドラーであった場合、ctx.prev.result はテンプレートの評価結果を表し、パイプライン内の最初の関数に使用可能になります。

前のオペレーションが最初の関数であった場合、ctx.prev.result は最初の関数レスポンスハンドラーの評価結果を表し、パイプライン内の 2 番目の関数に使用可能になります。

前のオペレーションが最後の関数であった場合、ctx.prev.result は最後の関数の評価結果を表し、パイプラインリゾルバーのレスポンスハンドラーに使用可能になります。

info

GraphQL リクエストに関する情報を含むオブジェクト。このフィールドの構造については、「情報」を参照してください。

アイデンティティ

`identity` セクションには、呼び出し元に関する情報が含まれています。このセクションのシェイプは、使用する AWS AppSync API の認可タイプによって異なります。

AWS AppSync セキュリティオプションの詳細については、「認可と認証」を参照してください。

`API_KEY` authorization

`identity` フィールドは入力されていません。

`AWS_LAMBDA` authorization

`identity` の形式は次のとおりです。

```
type AppSyncIdentityLambda = {
  resolverContext: any;
};
```

`identity`には、リクエストを承認する Lambda 関数によって返されるのと同じ `resolverContext` コンテンツを含む `resolverContext` キーが含まれています。

`AWS_IAM` authorization

`identity` の形式は次のとおりです。

```
type AppSyncIdentityIAM = {
  accountId: string;
  cognitoIdentityPoolId: string;
  cognitoIdentityId: string;
  sourceIp: string[];
  username: string;
  userArn: string;
  cognitoIdentityAuthType: string;
  cognitoIdentityAuthProvider: string;
};
```

`AMAZON_COGNITO_USER_POOLS` authorization

`identity` の形式は次のとおりです。

```
type AppSyncIdentityCognito = {
  sourceIp: string[];
```

```
  username: string;
  groups: string[] | null;
  sub: string;
  issuer: string;
  claims: any;
  defaultAuthStrategy: string;
};
```

各フィールドは次のように定義されています。

accountId

発信者の AWS アカウント ID。

claims

ユーザーが持っているクレーム。

cognitoIdentityAuthType

ID タイプに基づいて認証済みまたは未認証のいずれか。

cognitoIdentityAuthProvider

リクエストの署名に使用された認証情報の取得先となる外部 ID プロバイダ情報のカンマ区切りリスト。

cognitoIdentityId

リクエストを行う呼び出し元の Amazon Cognito 認証 ID。

cognitoIdentityPoolId

呼び出し元に関連付けられている Amazon Cognito ID プール。

defaultAuthStrategy

この呼び出し元のデフォルトの認可方法 (ALLOWまたはDENY)。

issuer

トークン発行者。

sourceIp

が AWS AppSync 受信する発信者の送信元 IP アドレス。リクエストに `x-forwarded-for` ヘッダーが含まれていない場合、ソース IP の値には TCP 接続からの 1 つの IP アドレスのみが含ま

れます。リクエストに x-forwarded-for ヘッダーが含まれている場合、ソース IP は、TCP 接続からの IP アドレスと x-forwarded-for ヘッダーからの IP アドレスのリストです。

sub

認証されたユーザーの UUID。

user

IAM ユーザー。

userArn

IAM ユーザーの Amazon リソースネーム (ARN)。

username

認証されたユーザーのユーザー名です。AMAZON_COGNITO_USER_POOLS 認可の場合、username の値は cognito:username 属性の値です。AWS_IAM 認可の場合、ユーザー名の値は AWS ユーザープリンシパルの値です。Amazon Cognito アイデンティティプールから発行された認証情報による IAM 認可を使用されている場合は、cognitoIdentityIdの使用をお勧めします。

リクエストヘッダーへのアクセス

AWS AppSync は、クライアントからのカスタムヘッダーの受け渡しと、 を使用した GraphQL リゾルバーでのカスタムヘッダーへのアクセスをサポートしていますctx.request.headers。データソースへデータの挿入や認可チェックなどのアクションでそのヘッダーの値を使用できます。次の例のようにコマンドラインから API キーで $curl を使用して、1 つまたは複数のリクエストヘッダーを使用できます。

単一ヘッダーの例

次のように、custom のヘッダーに nadia の値を設定しているとします。

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:nadia" -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}"}' https://<ENDPOINT>/graphql
```

この値には ctx.request.headers.custom を使用してアクセスできます。たとえば、DynamoDB に対する VTL は次のようになります。

```
"custom": util.dynamodb.toDynamoDB(ctx.request.headers.custom)
```

複数ヘッダーの例

1 つのリクエストで複数のヘッダーを渡して、リゾルバーのハンドラーでそれらのヘッダーにアクセスすることもできます。たとえば、次のように custom ヘッダーに 2 つの値が設定されるとします。

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:bailey" -H "custom:nadia"
 -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo
\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}"}'
 https://<ENDPOINT>/graphql
```

それらのヘッダーに配列としてアクセスできます (例: `ctx.request.headers.custom[1]`)。

> **Note**
>
> AWS AppSync は の Cookie ヘッダーを公開しませんctx.request.headers。

リクエストカスタムドメイン名にアクセスする

AWS AppSync は、APIs の GraphQL およびリアルタイムエンドポイントにアクセスするために使用できるカスタムドメインの設定をサポートしています。カスタムドメイン名を使用してリクエストを行う場合は、`ctx.request.domainName` を使用してドメイン名を取得できます。

デフォルトの GraphQL エンドポイントドメイン名を使用する場合、値は `null` です。

情報

`info` セクションには、GraphQL リクエストに関する情報が含まれています。このセクションには、次の形式が含まれます。

```
type Info = {
  fieldName: string;
  parentTypeName: string;
  variables: any;
  selectionSetList: string[];
  selectionSetGraphQL: string;
};
```

各フィールドは次のように定義されています。

`fieldName`

現在解決中のフィールドの名前。

`parentTypeName`

現在解決中のフィールドの親タイプの名前。

`variables`

GraphQLリクエストに渡されるすべての変数を保持するマップ。

`selectionSetList`

GraphQL 選択セット内のフィールドのリスト表現。エイリアスされたフィールドは、フィールド名ではなく、エイリアス名によってのみ参照されます。次の例で詳細を示します。

`selectionSetGraphQL`

選択セットの文字列表現。GraphQL スキーマ定義言語 (SDL) としてフォーマットされます。フラグメントは選択セットにマージされませんが、次の例に示すように、インラインフラグメントは保持されます。

> **Note**
>
> `JSON.stringify` では文字列のシリアル化には `selectionSetGraphQL` および `selectionSetList` は含まれません。これらのプロパティは直接参照する必要があります。

たとえば、次のクエリの `getPost` フィールドを解決する場合:

```
query {
  getPost(id: $postId) {
    postId
    title
    secondTitle: title
    content
    author(id: $authorId) {
      authorId
      name
```

```
    }
    secondAuthor(id: "789") {
      authorId
    }
    ... on Post {
      inlineFrag: comments: {
        id
      }
    }
    ... postFrag
  }
}

fragment postFrag on Post {
  postFrag: comments: {
    id
  }
}
```

ハンドラーを処理する際に使用可能な完全な ctx.info 変数は次のようになります。

```
{
  "fieldName": "getPost",
  "parentTypeName": "Query",
  "variables": {
    "postId": "123",
    "authorId": "456"
  },
  "selectionSetList": [
    "postId",
    "title",
    "secondTitle"
    "content",
    "author",
    "author/authorId",
    "author/name",
    "secondAuthor",
    "secondAuthor/authorId",
    "inlineFragComments",
    "inlineFragComments/id",
    "postFragComments",
    "postFragComments/id"
  ],
```

```
  "selectionSetGraphQL": "{\n  getPost(id: $postId) {\n    postId\n    title\n
 secondTitle: title\n    content\n    author(id: $authorId) {\n      authorId\n
 name\n    }\n    secondAuthor(id: \"789\") {\n      authorId\n    }\n    ... on Post
 {\n      inlineFrag: comments {\n        id\n      }\n    }\n    ... postFrag\n  }\n}"
}
```

selectionSetList は現在のタイプに属するフィールドのみを公開します。現在のタイプがインタフェースまたは共用体の場合、そのインタフェースに属する選択されたフィールドのみが公開されます。例として、次のスキーマがあるとします。

```
type Query {
    node(id: ID!): Node
}

interface Node {
    id: ID
}

type Post implements Node {
    id: ID
    title: String
    author: String
}

type Blog implements Node {
    id: ID
    title: String
    category: String
}
```

そして次のクエリがあります。

```
query {
    node(id: "post1") {
        id
        ... on Post {
            title
        }

        ... on Blog {
            title
        }
```

```
    }
}
```

Query.node フィールド解決で ctx.info.selectionSetList が呼び出されると、id のみが公開されます。

```
"selectionSetList": [
    "id"
]
```

リゾルバーと関数のAWS AppSync JavaScript ランタイム機能

APPSYNC_JS ランタイム環境には ECMAScript (ES) バージョン 6.0 と同様の機能が備わっています。一部の機能をサポートし、ES 仕様に含まれない追加のメソッド (ユーティリティ) も提供しています。次のトピックでは、サポートされるすべての言語機能の一覧を示します。

- サポートされているランタイム機能 - サポートされているコア機能、プリミティブオブジェクト、組み込みオブジェクトと関数などの詳細について説明します。
- 組み込みユーティリティ - 変数には、データの操作を容易にする一般的なユーティリティメソッドが含まれています。特に指定されていない限り、すべてのユーティリティでは UTF-8 文字セットが使用されます。
- 組み込みモジュール - 組み込みモジュールが JavaScript リゾルバーと関数の記述にどのように役立つかについて説明します。
- ランタイムユーティリティ - ランタイムライブラリには、リゾルバーと関数のランタイムプロパティを制御または変更するためのユーティリティが用意されています。
- util.time のタイムヘルパー - util.time 変数には、タイムスタンプの生成、日時形式間の変換、および日時文字列の解析に役立つ日時メソッドが含まれています。日時形式の構文は DateTimeFormatter に基づいています。詳細については、DateTimeFormatter を参照してください。
- util.dynamodb のDynamoDB ヘルパー - util.dynamodb には、Amazon DynamoDB に対するデータの読み書きを容易にするヘルパーメソッド (自動型マッピングやフォーマットなど) が含まれています。
- util.http の HTTP ヘルパー - util.http ユーティリティには、HTTP リクエストパラメータの管理やレスポンスヘッダーの追加に使用できるヘルパーメソッドが用意されています。
- util.transform の変換ヘルパー - util.transform には、データソースに対して複雑な操作を簡単に実行できるヘルパーメソッドが含まれています。

- util.str の文字列ヘルパー - util.str には、一般的な文字列オペレーションに役立つメソッドが含まれています。
- 拡張機能 - 拡張機能には、リゾルバー内で追加のアクションを行うためのメソッドのセットが含まれています。
- util.xml の XML ヘルパー - util.xml には、XML 文字列変換に役立つメソッドが含まれています。

> **Note**
>
> 現在、このリファレンスはランタイムバージョン 1.0.0 にのみ適用されます。

サポートされているランタイム機能

以下のセクションでは、APPSYNC_JS ランタイムでサポートされる機能セットについて説明します。

主要機能

次の主要機能がサポートされています。

Types

次のタイプの BGP がサポートされています。

- 数字
- 文字列
- ブール値
- objects
- 配列
- 関数

Operators

次のような演算子がサポートされています。

- 標準の数学演算子 (+、-、/、%、* など)

- Null 合体演算子 (??)
- オプションのチェーニング (?.)
- ビット演算子
- void および typeof 演算子
- スプレッド演算子 (...)

以下は、サポートされていない演算子です。

- 単項演算子 (++、--、および ~)
- inoperator

> **Note**
>
> Object.hasOwn 演算子を使用して、指定したプロパティが指定されたオブジェクト内にあるかどうかを調べます。

Statements

次のステートメントがサポートされています。

- const
- let
- var
- break
- else
- for-in
- for-of
- if
- return
- switch
- スプレッド構文

ただし、以下はサポートしていません。

- `catch`
- `continue`
- `do-while`
- `finally`
- `for(initialization; condition; afterthought)`

> **Note**
>
> ただし、サポートされている `for-in` および `for-of` 式は例外です。

- `throw`
- `try`
- `while`
- ラベル付きステートメント

Literals

以下の ES 6 テンプレートリテラルがサポートされています。

- 複数行文字列
- 式の補間
- ネストテンプレート

Functions

以下の構文がサポートされています。

- 関数宣言はサポートされています。
- ES 6 アロー関数はサポートされています。
- ES 6 REST パラメータ構文はサポートされています。

Strict mode

関数は Strict モードで動作するため、関数コードに `use_strict` ステートメントを追加する必要はありません。これは変更できません。

プリミティブオブジェクト

以下の ES プリミティブオブジェクトがサポートされています。

Object

以下のオブジェクトがサポートされています。

- `Object.assign()`
- `Object.entries()`
- `Object.hasOwn()`
- `Object.keys()`
- `Object.values()`
- `delete`

String

以下の文字列がサポートされています。

- `String.prototype.length()`
- `String.prototype.charAt()`
- `String.prototype.concat()`
- `String.prototype.endsWith()`
- `String.prototype.indexOf()`
- `String.prototype.lastIndexOf()`
- `String.raw()`
- `String.prototype.replace()`

> **Note**
>
> 正規表現はサポートされていません。
> ただし、Java スタイルの正規表現コンストラクトは、指定されたパラメータでサポートされています。詳細については、「パターン」を参照してください。

- `String.prototype.replaceAll()`

> **Note**
>
> 正規表現はサポートされていません。
> ただし、Java スタイルの正規表現コンストラクトは、指定されたパラメータでサポートされています。詳細については、「パターン」を参照してください。

- `String.prototype.slice()`
- `String.prototype.split()`
- `String.prototype.startsWith()`
- `String.prototype.toLowerCase()`
- `String.prototype.toUpperCase()`
- `String.prototype.trim()`
- `String.prototype.trimEnd()`
- `String.prototype.trimStart()`

Number

次の番号がサポートされています。

- `Number.isFinite`
- `Number.isNaN`

組み込みオブジェクトと関数

以下の ES 5.1 グローバル関数がサポートされています。

Math

次の Math 関数はサポートされていません。

- `Math.random()`
- `Math.min()`
- `Math.max()`
- `Math.round()`

- `Math.floor()`
- `Math.ceil()`

Array

以下の配列メソッドがサポートされています。

- `Array.prototype.length`
- `Array.prototype.concat()`
- `Array.prototype.fill()`
- `Array.prototype.flat()`
- `Array.prototype.indexOf()`
- `Array.prototype.join()`
- `Array.prototype.lastIndexOf()`
- `Array.prototype.pop()`
- `Array.prototype.push()`
- `Array.prototype.reverse()`
- `Array.prototype.shift()`
- `Array.prototype.slice()`
- `Array.prototype.sort()`

  > **Note**
  >
  > `Array.prototype.sort()` は引数はサポートしていません。

- `Array.prototype.splice()`
- `Array.prototype.unshift()`
- `Array.prototype.forEach()`
- `Array.prototype.map()`
- `Array.prototype.flatMap()`
- `Array.prototype.filter()`
- `Array.prototype.reduce()`

- `Array.prototype.reduceRight()`
- `Array.prototype.find()`
- `Array.prototype.some()`
- `Array.prototype.every()`
- `Array.prototype.findIndex()`
- `Array.prototype.findLast()`
- `Array.prototype.findLastIndex()`
- `delete`

Console

コンソールオブジェクトはデバッグに使用できます。ライブクエリの実行中、コンソールのログ/エラーステートメントが Amazon CloudWatch Logs に送信されます (ロギングが有効になっている場合)。`evaluateCode` でのコード評価中、コマンドレスポンスにログステートメントが返されます。

- `console.error()`
- `console.log()`

Function

- `apply`、`bind` `call` メソッドはサポートされていません。
- 関数コンストラクタはサポートされていません。
- 関数を引数として渡すことはサポートされていません。
- 再帰的な関数呼び出しはサポートされていません。

JSON

以下の JSON メソッドがサポートされています。

- `JSON.parse()`

> **Note**
>
> 解析された文字列が有効な JSON でない場合は、空白の文字列を返します。

- `JSON.stringify()`

Promises

非同期プロセスはサポートされておらず、Promise もサポートされていません。

> **Note**
>
> ネットワークおよびファイルシステムへのアクセスは、 APPSYNC_JSランタイム内ではサポートされていません AWS AppSync。 は、 AWS AppSync リゾルバーまたは AWS AppSync 関数によって行われたリクエストに基づいて、すべての I/O オペレーション AWS AppSync を処理します。

Globals

以下のグローバル定数がサポートされています。

- `NaN`
- `Infinity`
- `undefined`
- `util`
- `extensions`
- `runtime`

エラーのタイプ

`throw` でエラーをスローすることはサポートされていません。`util.error()` 関数を使用するとエラーを返すことができます。`util.appendError` 関数を使用して GraphQL レスポンスにエラーを含めることができます。

詳細については、「エラー utils」を参照してください。

組み込みユーティリティ

`util` 変数には、データの操作を容易にする一般的なユーティリティメソッドが含まれています。特に指定されていない限り、すべてのユーティリティでは UTF-8 文字セットが使用されます。

エンコーディング utils

エンコーディング utils リスト

`util.urlEncode(String)`

入力文字列を `application/x-www-form-urlencoded` でエンコードされた文字列として返します。

`util.urlDecode(String)`

`application/x-www-form-urlencoded` でエンコードされた文字列をデコードして、エンコードされていない形式に戻します。

`util.base64Encode(string) : string`

入力を base64 でエンコードされた文字列にエンコードします。

`util.base64Decode(string) : string`

base64 エンコードされた文字列からデータをデコードします。

ID 生成 utils

ID 生成 utils リスト

`util.autoId()`

ランダムに生成された 128 ビットの UUID を返します。

`util.autoUlid()`

ランダムに生成された 128 ビットの ULID (辞書的にソート可能なユニバーサルユニーク識別子) を返します。

`util.autoKsuid()`

長さ 27 の文字列としてエンコードされた、ランダムに生成された 128 ビットの KSUID (K ソート可能な固有識別子) base62 を返します。

エラー utils

エラー utils リスト

`util.error(String, String?, Object?, Object?)`

カスタムエラーをスローします。これは、テンプレートがリクエストまたは呼び出し結果でエラーを検出した場合に、リクエストまたはレスポンスのマッピングテンプレートで使用できます。また、errorType フィールド、data フィールド、および errorInfo フィールドを指定できます。data の値は、GraphQL レスポンスの errors 内の該当する error ブロックに追加されます。

> **Note**
>
> data はクエリ選択セットに基づいてフィルタリングされます。errorInfo の値は、GraphQL レスポンスの errors 内の該当する error ブロックに追加されます。errorInfo はクエリ選択セットに基づいてフィルタリングされません。

`util.appendError(String, String?, Object?, Object?)`

カスタムエラーを追加します。これは、テンプレートがリクエストまたは呼び出し結果でエラーを検出した場合に、リクエストまたはレスポンスのマッピングテンプレートで使用できます。また、errorType フィールド、data フィールド、および errorInfo フィールドを指定できます。util.error(String, String?, Object?, Object?) とは異なり、テンプレートの評価は中断されないため、データを呼び出し元に返すことができます。data の値は、GraphQL レスポンスの errors 内の該当する error ブロックに追加されます。

> **Note**
>
> data はクエリ選択セットに基づいてフィルタリングされます。errorInfo の値は、GraphQL レスポンスの errors 内の該当する error ブロックに追加されます。errorInfo はクエリ選択セットに基づいてフィルタリングされません。

タイプとパターンマッチングの utils

タイプとパターンマッチングの utils リスト

`util.matches(String, String) : Boolean`

最初の引数で指定されたパターンが、2 番目の引数で指定されたデータと一致する場合は true を返します。パターンは正規表現である必要があります (例: `util.matches("a*b", "aaaaab")`)。この機能は Java の Pattern に基づいています。詳細については、Pattern を参照してください。

`util.authType()`

リクエストで使用されているマルチ認可タイプを表す文字列型 (String) の値を返します。「IAM 認可」、「ユーザープールの認可」、「オープン ID Connect 認可」、または「API キー認可」のいずれかを返します。

戻り値動作 utils

戻り値動作 utils リスト

`util.escapeJavaScript(String)`

入力文字列を JavasScript のエスケープした文字列として返します。

リゾルバー認可 utils

リゾルバー認可 utils リスト

`util.unauthorized()`

解決されるフィールドに対して `Unauthorized` をスローします。リクエストまたはレスポンスのマッピングテンプレートでこれを使用し、呼び出し元にそのフィールドの解決が許可されるかどうかを判別します。

ビルトインモジュール

モジュールは `APPSYNC_JS` ランタイムの一部であり、JavaScript リゾルバーと関数の作成に役立つユーティリティを提供します。サンプルと例については、aws-appsync-resolver-samples GitHub リポジトリを参照してください。

DynamoDB モジュール関数

DynamoDB モジュール関数を使用すると、DynamoDB データソースを操作する際のエクスペリエンスが向上します。タイプマッピングを追加しなくても、関数を使用して DynamoDB データソースに対してリクエストを行うことができます。

モジュールは `@aws-appsync/utils/dynamodb` を使用してインポートされます。

```
// Modules are imported using @aws-appsync/utils/dynamodb
import * as ddb from '@aws-appsync/utils/dynamodb';
```

関数

関数のリスト

`get<T>(payload: GetInput): DynamoDBGetItemRequest`

> **Tip**
>
> GetInput については、「the section called “入力”」を参照してください。

DynamoDB に GetItem リクエストを行うための `DynamoDBGetItemRequest` オブジェクトを生成します。

```
import { get } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 return get({ key: { id: ctx.args.id } });
}
```

`put<T>(payload): DynamoDBPutItemRequest`

DynamoDB に PutItem リクエストを行うための `DynamoDBPutItemRequest` オブジェクトを生成します。

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 return ddb.put({ key: { id: util.autoId() }, item: ctx.args });
```

```
}
```

remove<T>(payload): DynamoDBDeleteItemRequest

DynamoDB に DeleteItem リクエストを行うための DynamoDBDeleteItemRequest オブジェクトを生成します。

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 return ddb.remove({ key: { id: ctx.args.id } });
}
```

scan<T>(payload): DynamoDBScanRequest

DynamoDB に Scan リクエストを行うためのDynamoDBScanRequestを生成します。

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const { limit = 10, nextToken } = ctx.args;
 return ddb.scan({ limit, nextToken });
}
```

sync<T>(payload): DynamoDBSyncRequest

Sync リクエストを行う DynamoDBSyncRequest オブジェクトを生成します。リクエストは、前回のクエリ (デルタ更新) 以降に変更されたデータのみを受け取ります。リクエストは、バージョン管理された DynamoDB データソースに対してのみ行うことができます。

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const { limit = 10, nextToken, lastSync } = ctx.args;
 return ddb.sync({ limit, nextToken, lastSync });
}
```

update<T>(payload): DynamoDBUpdateItemRequest

DynamoDB に UpdateItem リクエストを行うための DynamoDBUpdateItemRequest オブジェクトを生成します。

オペレーション

オペレーションヘルパーを使用すると、更新中にデータの一部に対して特定のアクションを実行できます。はじめに、@aws-appsync/utils/dynamodb から operations をインポートしてください。

```
// Modules are imported using operations
import {operations} from '@aws-appsync/utils/dynamodb';
```

オペレーションリスト

add<T>(payload)

DynamoDB を更新するときに新しい属性項目を追加するヘルパー関数。

例

ID 値を使用して既存の DynamoDB 項目に住所 (番地、市区町村、郵便番号) を追加するには

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const updateObj = {
  address: operations.add({
   street1: '123 Main St',
   city: 'New York',
   zip: '10001',
  }),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

append <T>(payload)

DynamoDB の既存のリストにペイロードを追加するヘルパー関数。

例

更新中に新しく追加されたフレンド ID (newFriendIds) を既存のフレンドリスト (friendsIds) に追加するには

```
import { update, operations } from '@aws-appsync/utils/dynamodb';
```

```
export function request(ctx) {
 const newFriendIds = [101, 104, 111];
 const updateObj = {
  friendsIds: operations.append(newFriendIds),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

decrement (by?)

DynamoDB を更新するときに項目内の既存の属性値をデクリメントするヘルパー関数。

例

フレンドカウンター (friendsCount) を 10 ずつ減らすには

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const updateObj = {
  friendsCount: operations.decrement(10),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

increment (by?)

DynamoDB を更新するときに項目内の既存の属性値をインクリメントするヘルパー関数。

例

フレンドカウンター (friendsCount) を 10 ずつ増やすには:

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const updateObj = {
  friendsCount: operations.increment(10),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

prepend <T>(payload)

DynamoDB の既存のリストの前に追加するヘルパー関数。

例

更新時に、新しく追加されたフレンド ID (newFriendIds) を既存のフレンドリスト (friendsIds) の先頭に追加するには

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const newFriendIds = [101, 104, 111];
 const updateObj = {
  friendsIds: operations.prepend(newFriendIds),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

replace <T>(payload)

DynamoDB の項目を更新するときに既存の属性を置き換えるヘルパー関数。これは、ペイロードのキーだけでなく、属性のオブジェクトまたはサブオブジェクト全体を更新したい場合に便利です。

例

info オブジェクト内の住所 (番地、市区町村、郵便番号) を置き換えるには

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const updateObj = {
  info: {
   address: operations.replace({
    street1: '123 Main St',
    city: 'New York',
    zip: '10001',
   }),
  },
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

updateListItem <T>(payload, index)

リスト内の項目を置き換えるヘルパー関数。

例

更新 (newFriendIds) のスコープで、この例では friendsIds を使用してリスト (updateListItem) の 2 番目の項目 (index: 1、new ID:102) と 3 番目の項目 (index: 2、new ID:112) の ID 値を更新しました。

```
import { update, operations as ops } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const newFriendIds = [
  ops.updateListItem('102', 1), ops.updateListItem('112', 2)
 ];
 const updateObj = { friendsIds: newFriendIds };
 return update({ key: { id: 1 }, update: updateObj });
}
```

入力

入力リスト

Type GetInput<T>

```
GetInput<T>: {
    consistentRead?: boolean;
    key: DynamoDBKey<T>;
}
```

タイプ宣言

- consistentRead?: boolean (オプション)

  DynamoDB で強力な整合性のある読み込みを実行するかどうかを指定するオプションのブール値。

- key: DynamoDBKey<T> (必須)

  DynamoDB の項目のキーを指定する必須パラメータ。DynamoDB の項目には、単一のハッシュキーとソートキーが含まれています。

Type PutInput<T>

```
PutInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T> | null;
    customPartitionKey?: string;
    item: Partial<T>;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
}
```

タイプ宣言

- `_version?: number` (オプション)
- `condition?: DynamoDBFilterObject<T> | null` (オプション)

  DynamoDB テーブルにオブジェクトを格納する際、オプションで条件式を指定することができます。この条件式は、操作を実行する前にすでに DynamoDB に格納されているオブジェクトの状態に基づいて、リクエストを成功させるかどうかを制御します。
- `customPartitionKey?: string` (オプション)

  有効にすると、この文字列値によって、バージョニングが有効になっているときにデルタ同期テーブルが使用する ds_sk および ds_pk レコードの形式が変更されます。有効にすると、populateIndexFields エントリの処理も有効になります。
- `item: Partial<T>` (必須)

  DynamoDB に配置される項目の残りの属性です。
- `key: DynamoDBKey<T>` (必須)

  Put が実行される DynamoDB 内の項目のキーを指定する必須パラメータ。DynamoDB の項目には、単一のハッシュキーとソートキーが含まれています。
- `populateIndexFields?: boolean` (オプション)

  ブール値で、customPartitionKey と一緒に有効にすると、差分同期テーブル、具体的には gsi_ds_pk と gsi_ds_sk 列のレコードごとに新しいエントリが作成されます。詳細については、「AWS AppSync デベロッパーガイド」の「構成項目の配信」を参照してください。

Type QueryInput<T>

```
QueryInput<T>: ScanInput<T> & {
    query: DynamoDBKeyCondition<Required<T>>;
```

```
}
```

タイプ宣言

- query: DynamoDBKeyCondition<Required<T>> (必須)

  クエリする項目を記述するキー条件を指定します。特定のインデックスでは、パーティションキーの条件は等しく、ソートキーは比較または beginsWith (文字列の場合) でなければなりません。パーティションキーとソートキーでは、数値型と文字列型のみがサポートされています。

  例

  以下の User タイプを考えてみてください。

  ```
  type User = {
    id: string;
    name: string;
    age: number;
    isVerified: boolean;
    friendsIds: string[]
  }
  ```

  このクエリにはid、name、age のフィールドのみを含めることができます。

  ```
  const query: QueryInput<User> = {
      name: { eq: 'John' },
      age: { gt: 20 },
  }
  ```

Type RemoveInput<T>

```
RemoveInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T>;
    customPartitionKey?: string;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
}
```

タイプ宣言

- `_version?: number` (オプション)
- `condition?: DynamoDBFilterObject<T>` (オプション)

  DynamoDB テーブルにオブジェクトを格納する際、オプションで条件式を指定することができます。この条件式は、操作を実行する前にすでに DynamoDB に格納されているオブジェクトの状態に基づいて、リクエストを成功させるかどうかを制御します。

  例

  次の例は、ドキュメントの所有者がリクエストを行ったユーザーと一致する場合にのみ操作が成功することを許可する条件を含む DeleteItem 式です。

```
type Task = {
  id: string;
  title: string;
  description: string;
  owner: string;
  isComplete: boolean;
}
const condition: DynamoDBFilterObject<Task> = {
  owner: { eq: 'XXXXXXXXXXXXXXXX' },
}

remove<Task>({
   key: {
     id: 'XXXXXXXXXXXXXXXX',
  },
  condition,
});
```

- `customPartitionKey?: string` (オプション)

  有効にすると、customPartitionKey 値によって、バージョニングが有効になっているときにデルタ同期テーブルが使用する ds_sk および ds_pk レコードの形式が変更されます。有効にすると、populateIndexFields エントリの処理も有効になります。

- `key: DynamoDBKey<T>` (必須)

  削除する DynamoDB 内の項目のキーを指定する必須パラメータ。DynamoDB の項目には、単一のハッシュキーとソートキーが含まれています。

  例

User がユーザー id のハッシュキーしか持っていない場合、キーは次のようになります。

```
type User = {
 id: number
 name: string
 age: number
 isVerified: boolean
}
const key: DynamoDBKey<User> = {
 id: 1,
}
```

テーブルユーザーがハッシュキー (id) とソートキー (name) を持っている場合、キーは以下のようになります。

```
type User = {
 id: number
 name: string
 age: number
 isVerified: boolean
 friendsIds: string[]
}

const key: DynamoDBKey<User> = {
 id: 1,
 name: 'XXXXXXXXXX',
}
```

- populateIndexFields?: boolean (オプション)

  ブール値で、customPartitionKey と一緒に有効にすると、差分同期テーブル、具体的には gsi_ds_pk と gsi_ds_sk 列のレコードごとに新しいエントリが作成されます。

Type ScanInput<T>

```
ScanInput<T>: {
    consistentRead?: boolean | null;
    filter?: DynamoDBFilterObject<T> | null;
    index?: string | null;
    limit?: number | null;
    nextToken?: string | null;
    scanIndexForward?: boolean | null;
```

```
    segment?: number;
    select?: DynamoDBSelectAttributes;
    totalSegments?: number;
}
```

タイプ宣言

- `consistentRead?: boolean | null` (オプション)

  DynamoDB のクエリ時に一貫性のあるリードを示すオプションのブール値。デフォルト値は `false` です。

- `filter?: DynamoDBFilterObject<T> | null` (オプション)

  テーブルから結果を取得した後に結果に適用するオプションのフィルター。

- `index?: string | null` (オプション)

  スキャンするインデックスの名前。

- `limit?: number | null` (オプション)

  返される結果の最大数。

- `nextToken?: string | null` (オプション)

  前のクエリを継続するためのオプションのページ分割トークンです。これは前のクエリから取得されます。

- `scanIndexForward?: boolean | null` (オプション)

  クエリを昇順にするか、降順にするかを示すオプションのブール値。デフォルトではこの値は `true` に設定されます。

- `segment?: number` (オプション)
- `select?: DynamoDBSelectAttributes` (オプション)

  DynamoDB から返される属性。デフォルトでは、 AWS AppSync DynamoDB リゾルバーはインデックスに射影された属性のみを返します。サポートされている値には以下があります。

  - `ALL_ATTRIBUTES`

    指定されたテーブルまたはインデックスのすべての項目の属性を返します。ローカルセカンダリインデックスに対してクエリを実行する場合、DynamoDB は、親のテーブルからインデックスの項目に一致したすべての項目をフェッチします。インデックスがすべての項目の

属性を射影するように設定されている場合、すべてのデータはローカルセカンダリインデックスから取得されるため、フェッチは必要ありません。

- ALL_PROJECTED_ATTRIBUTES

  インデックスにプロジェクションされたすべての属性を取得します。インデックスがすべての属性を投射するように設定されている場合、この返り値は ALL_ATTRIBUTES を指定した場合と同等になります。

- SPECIFIC_ATTRIBUTES

  ProjectionExpression にリストされている属性のみを返します。この戻り値は、AttributesToGet の値を指定せずに ProjectionExpression を指定することと同じです。

- totalSegments?: number (オプション)

Type DynamoDBSyncInput<T>

```
DynamoDBSyncInput<T>: {
    basePartitionKey?: string;
    deltaIndexName?: string;
    filter?: DynamoDBFilterObject<T> | null;
    lastSync?: number;
    limit?: number | null;
    nextToken?: string | null;
}
```

タイプ宣言

- basePartitionKey?: string (オプション)

  Sync 操作を実行するときに使用するベーステーブルのパーティションキー。このフィールドにより、テーブルがカスタムパーティションキーを使用している場合に Sync 操作を実行できます。

- deltaIndexName?: string (オプション)

  Sync 操作に使用されるインデックス。このインデックスは、テーブルがカスタムパーティションキーを使用する場合に、デルタストアテーブル全体で Sync 操作を有効にするために必要です。Sync オペレーションは GSI (gsi_ds_pk と gsi_ds_sk で作成) で実行されます。

- filter?: DynamoDBFilterObject<T> | null (オプション)

  テーブルから結果を取得した後に結果に適用するオプションのフィルター。

- lastSync?: number (オプション)

  最後に成功した Sync オペレーションが開始されたエポックミリ秒単位の時刻。指定すると、lastSync 以降に変更された項目のみが返されます。このフィールドはオプションです。最初の Sync オペレーションからすべてのページを取得した後にのみ入力する必要があります。省略すると、ベーステーブルの結果が返されます。それ以外の場合は、差分テーブルの結果が返されます。

- limit?: number | null (オプション)

  一度に評価する項目の最大数です。省略した場合、デフォルトの制限は 100 項目に設定されます。このフィールドの最大値は 1000 項目です。

- nextToken?: string | null (オプション)

Type DynamoDBUpdateInput<T>

```
DynamoDBUpdateInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T>;
    customPartitionKey?: string;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
    update: DynamoDBUpdateObject<T>;
}
```

タイプ宣言

- _version?: number (オプション)
- condition?: DynamoDBFilterObject<T> (オプション)

  DynamoDB でオブジェクトを更新する場合、オプションで条件式を指定することができます。この条件式は、操作を実行する前にすでに DynamoDB にあるオブジェクトの状態に基づいて、リクエストを成功させるかどうかを制御します。

- customPartitionKey?: string (オプション)

  有効にすると、customPartitionKey 値によって、バージョニングが有効になっているときにデルタ同期テーブルが使用する ds_sk および ds_pk レコードの形式が変更されます。有効にすると、populateIndexFields エントリの処理も有効になります。

- key: DynamoDBKey<T> (必須)

更新中の DynamoDB 内の項目のキーを指定する必須パラメータ。DynamoDB の項目には、単一のハッシュキーとソートキーが含まれています。

- `populateIndexFields?: boolean` (オプション)

  ブール値で、`customPartitionKey` と一緒に有効にすると、差分同期テーブル、具体的には `gsi_ds_pk` と `gsi_ds_sk` 列のレコードごとに新しいエントリが作成されます。

- `update: DynamoDBUpdateObject<T>`

  更新する属性とその新しい値を指定するオブジェクト。更新オブジェクトは `add`、`remove`、`replace`、`increment`、`decrement`、`append`、`prepend`、`updateListItem` と共に使用できます。

Amazon RDS モジュール関数

Amazon RDS モジュール関数を使用すると、Amazon RDS Data API で設定されたデータベースを操作する際のエクスペリエンスが向上します。モジュールは `@aws-appsync/utils/rds` を使用してインポートされます。

```
import * as rds from '@aws-appsync/utils/rds';
```

関数は個別にインポートすることもできます。例えば、以下のインポートでは `sql` を使用します。

```
import { sql } from '@aws-appsync/utils/rds';
```

関数

AWS AppSync RDS モジュールのユーティリティヘルパーを使用して、データベースとやり取りできます。

Select

`select` ユーティリティは、リレーショナルデータベースにクエリを実行する `SELECT` ステートメントを作成します。

基本的な使用法

基本的な形式では、クエリを実行するテーブルを指定できます。

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';
```

```
export function request(ctx) {

    // Generates statement:
    // "SELECT * FROM "persons"
    return createPgStatement(select({table: 'persons'}));
}
```

なお、テーブル識別子でスキーマを指定することもできます。

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // SELECT * FROM "private"."persons"
    return createPgStatement(select({table: 'private.persons'}));
}
```

列の指定

columns プロパティで列を指定できます。これを値に設定しない場合、デフォルトでは * になります。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name']
    }));
}
```

また、列のテーブルも指定できます。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "persons"."name"
    // FROM "persons"
```

```
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'persons.name']
    }));
}
```

LIMIT と OFFSET

limit と offset をクエリに適用できます。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // LIMIT :limit
    // OFFSET :offset
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        limit: 10,
        offset: 40
    }));
}
```

ORDER BY

結果は orderBy プロパティを使用してソートできます。列とオプションの dir プロパティを指定するオブジェクトの配列を指定します。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name" FROM "persons"
    // ORDER BY "name", "id" DESC
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        orderBy: [{column: 'name'}, {column: 'id', dir: 'DESC'}]
    }));
}
```

フィルター

特殊条件オブジェクトを使用してフィルターを構築できます。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}}
    }));
}
```

以下のフィルターを組み合わせることもできます。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME and "id" > :ID
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}, id: {gt: 10}}
    }));
}
```

また、OR ステートメントも作成できます。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME OR "id" > :ID
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: { or: [
            { name: { eq: 'Stephane'} },
```

```
            { id: { gt: 10 } }
        ]}
    }));
}
```

not を使用して条件を無効にすることもできます。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE NOT ("name" = :NAME AND "id" > :ID)
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: { not: [
            { name: { eq: 'Stephane'} },
            { id: { gt: 10 } }
        ]}
    }));
}
```

以下の演算子を使用して、値を比較することもできます。

演算子	説明	可能な値型
eq	Equal	number, string, boolean
ne	Not equal	number, string, boolean
le	Less than or equal	number, string
lt	Less than	number, string
ge	Greater than or equal	number, string
gt	Greater than	number, string
contains	Like	string
notContains	Not like	string

beginsWith	Starts with prefix	string
between	Between two values	number, string
attributeExists	The attribute is not null	number, string, boolean
size	checks the length of the element	string

Insert

`insert` ユーティリティによって、INSERT オペレーションを使用してデータベースに単一行の項目を簡単に挿入できます。

単一項目の挿入

項目を挿入するには、テーブルを指定して、値のオブジェクトを渡します。オブジェクトキーはテーブルの列にマッピングされます。列名は自動的に エスケープされ、値は変数マップを使用してデータベースに送信されます。

```
import { insert, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({ table: 'persons', values });

    // Generates statement:
    // INSERT INTO `persons`(`name`)
    // VALUES(:NAME)
    return createMySQLStatement(insertStatement)
}
```

MySQL ユースケース

`insert` の後に `select` を組み合わせて、挿入した行を取得できます。

```
import { insert, select, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({  table: 'persons', values });
```

```
    const selectStatement = select({
        table: 'persons',
        columns: '*',
        where: { id: { eq: values.id } },
        limit: 1,
    });

    // Generates statement:
    // INSERT INTO `persons`(`name`)
    // VALUES(:NAME)
    // and
    // SELECT *
    // FROM `persons`
    // WHERE `id` = :ID
    return createMySQLStatement(insertStatement, selectStatement)
}
```

Postgres のユースケース

Postgres では、returning を使用して、挿入した行からデータを取得できます。* または列名の配列が受け入れられます。

```
import { insert, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({
        table: 'persons',
        values,
        returning: '*'
    });

    // Generates statement:
    // INSERT INTO "persons"("name")
    // VALUES(:NAME)
    // RETURNING *
    return createPgStatement(insertStatement)
}
```

更新

update ユーティリティでは、既存の行を更新できます。条件オブジェクトを使用すると、条件を満たすすべての行の指定された列に変更を適用できます。例えば、このミューテーションを可能にする

スキーマがあるとします。Person の name を 3 の id 値で更新したいのですが、これは 2000 年以降にそれら (known_since) がわかっている場合に限ります。

```
mutation Update {
    updatePerson(
        input: {id: 3, name: "Jon"},
        condition: {known_since: {ge: "2000"}}
    ) {
    id
    name
  }
}
```

更新リゾルバーは以下のようになります。

```
import { update, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: { id, ...values }, condition } = ctx.args;
    const where = {
        ...condition,
        id: { eq: id },
    };
    const updateStatement = update({
        table: 'persons',
        values,
        where,
        returning: ['id', 'name'],
    });

    // Generates statement:
    // UPDATE "persons"
    // SET "name" = :NAME, "birthday" = :BDAY, "country" = :COUNTRY
    // WHERE "id" = :ID
    // RETURNING "id", "name"
    return createPgStatement(updateStatement)
}
```

条件にチェックを追加して、3 と等しいプライマリキー id を持つ行だけが更新されるようにすることができます。同様に、Postgres inserts の場合も、returning を使用して変更されたデータを返すことができます。

削除

remove ユーティリティでは、既存の行を削除できます。条件オブジェクトは、条件を満たすすべての行で使用できます。なお、delete はJavaScriptの予約キーワードです。代わりに remove を使ってください。

```
import { remove, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: { id }, condition } = ctx.args;
    const where = { ...condition, id: { eq: id } };
    const deleteStatement = remove({
        table: 'persons',
        where,
        returning: ['id', 'name'],
    });

    // Generates statement:
    // DELETE "persons"
    // WHERE "id" = :ID
    // RETURNING "id", "name"
    return createPgStatement(updateStatement)
}
```

キャスティング

ステートメントで使用する正しいオブジェクト型に関して、さらに特異度が必要となる場合もあるでしょう。指定された型ヒントを使用してパラメータのタイプを指定できます。 は Data API と同じ型ヒント AWS AppSync をサポートします。モジュールの typeHint関数を使用してパラメータを AWS AppSync rdsキャストできます。

次の例では、JSON オブジェクトとしてキャストされた値として配列を送信できます。-> 演算子を使用して JSON 配列内にある index 2 の要素を取得します。

```
import { sql, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/
rds';

export function request(ctx) {
    const arr = ctx.args.list_of_ids
    const statement = sql`select ${typeHint.JSON(arr)}->2 as value`
    return createPgStatement(statement)
}
```

```
export function response(ctx) {
    return toJsonObject(ctx.result)[0][0].value
}
```

キャスティングは、DATE、TIME、TIMESTAMP の処理や比較を行うときにも役立ちます。

```
import { select, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const when = ctx.args.when
    const statement = select({
        table: 'persons',
        where: { createdAt : { gt: typeHint.DATETIME(when) } }
    })
    return createPgStatement(statement)
}
```

現在の日付と時刻を送信する方法について別の例を以下に示します。

```
import { sql, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const now = util.time.nowFormatted('YYYY-MM-dd HH:mm:ss')
    return createPgStatement(sql`select ${typeHint.TIMESTAMP(now)}`)
}
```

使用可能な型ヒント

- typeHint.DATE - 対応するパラメータが、DATE 型のオブジェクトとしてデータベースに送信されます。受け入れられる形式は YYYY-MM-DD です。
- typeHint.DECIMAL - 対応するパラメータが、DECIMAL 型のオブジェクトとしてデータベースに送信されます。
- typeHint.JSON - 対応するパラメータが、JSON 型のオブジェクトとしてデータベースに送信されます。
- typeHint.TIME - 対応する文字列パラメータ値が、TIME 型のオブジェクトとしてデータベースに送信されます。受け入れられる形式は HH:MM:SS[.FFF] です。
- typeHint.TIMESTAMP - 対応する文字列パラメータ値が、TIMESTAMP 型のオブジェクトとしてデータベースに送信されます。受け入れられる形式は YYYY-MM-DD HH:MM:SS[.FFF] です。

- typeHint.UUID - 対応する文字列パラメータ値が、UUID 型のオブジェクトとしてデータベースに送信されます。

ランタイムユーティリティ

runtime ライブラリには、リゾルバーと関数のランタイムプロパティを制御または変更するためのユーティリティが用意されています。

ランタイム utils リスト

runtime.earlyReturn(obj?: unknown, returnOptions?: {skipTo: 'END' | 'NEXT'}): never

この関数を呼び出すと、現在のコンテキストに応じて、現在のハンドラー、 AWS AppSync 関数、またはリゾルバー (ユニットまたはパイプラインリゾルバー) の実行が停止します。指定したオブジェクトが結果として返されます。

- AWS AppSync 関数リクエストハンドラーで呼び出されると、データソースとレスポンスハンドラーはスキップされ、次の関数リクエストハンドラー (または最後の AWS AppSync 関数の場合はパイプラインリゾルバーレスポンスハンドラー) が呼び出されます。
- AWS AppSync パイプラインリゾルバーリクエストハンドラーで呼び出されると、パイプラインの実行はスキップされ、パイプラインリゾルバーレスポンスハンドラーはすぐに呼び出されます。
- returnOptions が「END」にskipTo設定されている場合、パイプラインの実行はスキップされ、パイプラインリゾルバーレスポンスハンドラーはすぐに呼び出されます。
- returnOptions が「NEXT」にskipTo設定されている場合、関数の実行はスキップされ、次のパイプラインハンドラーが呼び出されます。

例

```
import { runtime } from '@aws-appsync/utils'

export function request(ctx) {
  runtime.earlyReturn({ hello: 'world' })
  // code below is not executed
  return ctx.args
}

// never called because request returned early
export function response(ctx) {
```

```
  return ctx.result
}
```

util.time の日時ヘルパー

`util.time` 変数には、タイムスタンプの生成、日時形式間の変換、および日時文字列の解析に役立つ日時メソッドが含まれています。日時形式の構文は Java の DateTimeFormatter に基づいています。詳細については、DateTimeFormatter を参照してください。

日時 utils リスト

`util.time.nowISO8601()`

ISO 8601 形式の UTC の文字列型表現を返します。

`util.time.nowEpochSeconds()`

エポック (1970-01-01T00:00:00Z) から現在までの秒数を返します。

`util.time.nowEpochMilliSeconds()`

エポック (1970-01-01T00:00:00Z) から現在までのミリ秒数を返します。

`util.time.nowFormatted(String)`

文字列型の入力引数で指定された形式を使用して、UTC での現在のタイムスタンプを返します。

`util.time.nowFormatted(String, String)`

文字列型の入力引数で指定された形式とタイムゾーンを使用して、そのタイムゾーンでの現在のタイムスタンプを返します。

`util.time.parseFormattedToEpochMilliSeconds(String, String)`

文字列として渡されたタイムスタンプとタイムゾーンを含む形式を解析し、エポックからのミリ秒単位でタイムスタンプを返します。

`util.time.parseFormattedToEpochMilliSeconds(String, String, String)`

文字列型として渡されたタイムスタンプ、形式、およびタイムゾーンを解析し、エポックからのミリ秒単位のタイムスタンプを返します。

`util.time.parseISO8601ToEpochMilliSeconds(String)`

文字列型として渡された ISO8601 形式のタイムスタンプを解析し、エポックからのミリ秒単位のタイムスタンプを返します。

util.time.epochMilliSecondsToSeconds(long)

エポックからのミリ秒単位のタイムスタンプを、エポックからの秒単位のタイムスタンプに変換します。

util.time.epochMilliSecondsToISO8601(long)

エポックからのミリ秒単位のタイムスタンプを、ISO8601 形式のタイムスタンプに変換します。

util.time.epochMilliSecondsToFormatted(long, String)

long として渡されたエポックからのミリ秒単位のタイムスタンプを、文字列型で指定された形式に合わせて UTC でのタイムスタンプに変換します。

util.time.epochMilliSecondsToFormatted(long, String, String)

long として渡されたエポックからのミリ秒単位のタイムスタンプを、文字列型で指定された形式とタイムゾーンに合わせて、そのタイムゾーンでのタイムスタンプに変換します。

util.dynamodb の DynamoDB ヘルパー

`util.dynamodb` には、Amazon DynamoDB に対するデータの読み書きを容易にするヘルパーメソッド (自動型マッピングやフォーマットなど) が含まれています。

toDynamoDB

toDynamoDB utils リスト

util.dynamodb.toDynamoDB(Object)

入力されたオブジェクトを適切な DynamoDB 表現形式に変換する DynamoDB 用の一般的なオブジェクト変換ツールです。このツールは、一部の型の表現方法に関して融通が利きません。例えば、セット型 ("SS"、"NS"、"BS") ではなくリスト型 ("L") が使用されます。これは、DynamoDB 属性値を記述するオブジェクトを返します。

文字列型の例

```
Input:      util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

数値型の例

```
Input:      util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

ブール型の例

```
Input:      util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

リスト型の例

```
Input:      util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

マップ型の例

```
Input:      util.dynamodb.toDynamoDB({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
                           { "S" : "boop" }
                       ]
                   }
               }
           }
```

toString utils

toString utils リスト

`util.dynamodb.toString(String)`

入力文字列を DynamoDB の文字列形式に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

`util.dynamodb.toStringSet(List<String>)`

文字列型のリスト型を DynamoDB の文字列セット形式に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

toNumber utils

toNumber utils リスト

`util.dynamodb.toNumber(Number)`

数値を DynamoDB の数値形式に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

`util.dynamodb.toNumberSet(List<Number>)`

数値のリストを DynamoDB の数値セット形式に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

toBinary utils

toBinary utils リスト

`util.dynamodb.toBinary(String)`

base64 文字列としてエンコードされたバイナリデータを DynamoDB のバイナリ形式に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

`util.dynamodb.toBinarySet(List<String>)`

base64 文字列としてエンコードされたバイナリデータのリストを DynamoDB のバイナリセット形式に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

toBoolean utils

toBoolean utils リスト

`util.dynamodb.toBoolean(Boolean)`

ブール値を DynamoDB の該当するブール形式に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

toNull utils

toNull utils リスト

`util.dynamodb.toNull()`

null を DynamoDB の null 形式で返します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      util.dynamodb.toNull()
Output:     { "NULL" : null }
```

toList utils

toList utils リスト

`util.dynamodb.toList(List)`

オブジェクトのリストを DynamoDB のリスト形式に変換します。リスト内の各項目も、該当する DynamoDB 形式に変換されます。このツールは、一部のネステッドオブジェクトの表現方法に関して融通が利きません。例えば、セット型 ("SS"、"NS"、"BS") ではなくリスト型 ("L") が使用されます。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

toMap utils

toMap utils リスト

`util.dynamodb.toMap(Map)`

マップを DynamoDB のマップ形式に変換します。マップ内の各値も、該当する DynamoDB 形式に変換されます。このツールは、一部のネステッドオブジェクトの表現方法に関して融通が利きません。例えば、セット型 ("SS"、"NS"、"BS") ではなくリスト型 ("L") が使用されます。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      util.dynamodb.toMap({ "foo": "bar", "baz" : 1234, "beep": [ "boop"] })
```

```
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
                           { "S" : "boop" }
                       ]
                   }
               }
           }
```

util.dynamodb.toMapValues(Map)

マップ内の各値を該当する DynamoDB 形式に変換して、マップのコピーを作成します。このツールは、一部のネステッドオブジェクトの表現方法に関して融通が利きません。例えば、セット型 ("SS"、"NS"、"BS") ではなくリスト型 ("L") が使用されます。

```
Input:      util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "foo"  : { "S" : "bar" },
               "baz"  : { "N" : 1234 },
               "beep" : {
                   "L" : [
                       { "S" : "boop" }
                   ]
               }
           }
```

> **Note**
>
> 注 : このツールは `util.dynamodb.toMap(Map)` と少し異なっていて、属性値全体ではなく DynamoDB の属性値の内容のみを返します。例えば、次の 2 つのステートメントはまったく同じです。
>
> ```
> util.dynamodb.toMapValues(<map>)
> util.dynamodb.toMap(<map>)("M")
> ```

S3Object utils

S3Object utils リスト

`util.dynamodb.toS3Object(String key, String bucket, String region)`

キー、バケット、およびリージョンを DynamoDB の S3 オブジェクト表現に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      util.dynamodb.toS3Object("foo", "bar", region = "baz")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo", \"bucket\" : \"bar", \"region
\" : \"baz" } }" }
```

`util.dynamodb.toS3Object(String key, String bucket, String region, String version)`

キー、バケット、リージョン、およびバージョン (省略可) を DynamoDB の S3 オブジェクト表現に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      util.dynamodb.toS3Object("foo", "bar", "baz", "beep")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region
\" : \"baz\", \"version\" = \"beep\" } }" }
```

`util.dynamodb.fromS3ObjectJson(String)`

DynamoDB の S3 オブジェクトの文字列値を受け入れて、キー、バケット、リージョン、およびバージョン (省略可) が含まれているマップを返します。

```
Input:      util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
 \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :
 "beep" }
```

util.http の HTTP ヘルパー

`util.http` ユーティリティには、HTTP リクエストパラメータの管理やレスポンスヘッダーの追加に使用できるヘルパーメソッドが用意されています。

util.http utils リスト

`util.http.copyHeaders(headers)`

以下の制限された HTTP ヘッダーを除き、マップからヘッダーをコピーします。

- transfer-encoding
- connection
- host
- expect
- keep-alive
- upgrade
- proxy-authenticate
- proxy-authorization
- te
- content-length

`util.http.addResponseHeader(String, Object)`

レスポンスの名前 (`String`) と値 (`Object`) を含むカスタムヘッダーを 1 つ追加します。以下の制限が適用されます。

- の制限付きヘッダーのリストに加えて`copyHeaders(headers)`、ヘッダー名は次のいずれかと一致することはできません。
  - Access-Control-Allow-Credentials
  - Access-Control-Allow-Origin
  - Access-Control-Expose-Headers
  - Access-Control-Max-Age
  - Access-Control-Allow-Methods
  - Access-Control-Allow-Headers
  - Vary
  - Content-Type
- ヘッダー名は、制限付きプレフィックス `x-amzn-`または で始めることはできません`x-amz-`。
- カスタムレスポンスヘッダーのサイズは 4 KB を超えることはできません。これにはヘッダー名と値が含まれます。

- 各レスポンスヘッダーは、GraphQL 操作ごとに 1 回定義する必要があります。ただし、同じ名前のカスタムヘッダーを複数回定義すると、最新の定義がレスポンスに表示されます。名前に関係なく、すべてのヘッダーがヘッダーサイズの上限に含まれます。
- 名前が空または制限されているヘッダー(String)や null のヘッダー(Object)は無視され、オペレーションのerrors出力に追加されるResponseHeaderErrorエラーが発生します。

```
export function request(ctx) {
  util.http.addResponseHeader('itemsCount', 7)
  util.http.addResponseHeader('render', ctx.args.render)
  return {}
}
```

util.http.addResponseHeaders(Map)

指定された名前(String)と値のマップ からのレスポンスに複数のレスポンスヘッダーを追加します(Object)。addResponseHeader(String, Object) メソッドにリストされているのと同じ制限が、このメソッドにも適用されます。

```
export function request(ctx) {
  const headers = {
    headerInt: 12,
    headerString: 'stringValue',
    headerObject: {
      field1: 7,
      field2: 'string'
    }
  }
  util.http.addResponseHeaders(headers)
  return {}
}
```

util.transform の変換ヘルパー

util.transform には、Amazon DynamoDB フィルター処理などの、データソースに対する複雑なオペレーションの実行を容易にするヘルパーメソッドが含まれています。

変換ヘルパー utils リスト

`util.transform.toDynamoDBFilterExpression(filterObject: DynamoDBFilterObject) : string`

Amazon DynamoDB で使用するために、入力文字列をフィルター式に変換します。`toDynamoDBFilterExpression` を使用して、組み込みモジュール関数と併用することをおすすめします。

`util.transform.toElasticsearchQueryDSL(object: OpenSearchQueryObject) : string`

指定された入力を同等の OpenSearch Query DSL 式に変換し、JSON 文字列として返します。

入力例

```
util.transform.toElasticsearchQueryDSL({
    "upvotes":{
        "ne":15,
        "range":[
            10,
            20
        ]
    },
    "title":{
        "eq":"hihihi",
        "wildcard":"h*i"
    }
  })
```

出力例:

```
{
    "bool":{
      "must":[
          {
            "bool":{
              "must":[
                  {
                    "bool":{
                      "must_not":{
                        "term":{
```

```
                        "upvotes":15
                      }
                    }
                  }
                },
                {
                  "range":{
                    "upvotes":{
                      "gte":10,
                      "lte":20
                    }
                  }
                }
              ]
            }
          },
          {
            "bool":{
              "must":[
                  {
                    "term":{
                      "title":"hihihi"
                    }
                  },
                  {
                  "wildcard":{
                      "title":"h*i"
                    }
                  }
              ]
            }
          }
      ]
    }
}
```

> **Note**
>
> デフォルトの演算子は AND であると仮定されます。

util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter

Map 入力オブジェクトを SubscriptionFilter 式オブジェクトに変換します。util.transform.toSubscriptionFilter メソッドはextensions.setSubscriptionFilter() 拡張子への入力として使用されます。詳細については、「拡張子の使用」を参照してください。

> **Note**
>
> パラメータと return ステートメントは以下のとおりです。
> パラメータ
>
> - objFilter: SubscriptionFilterObject
>
>   SubscriptionFilter式オブジェクトに変換された Map 入力オブジェクト。
> - ignoredFields: SubscriptionFilterExcludeKeysType (オプション)
>
>   最初のオブジェクトのフィールド名は無視されます。
> - rules: SubscriptionFilterRuleObject (オプション)
>
>   SubscriptionFilter 式オブジェクトを作成する際に含める、厳格なルールが適用された Map 入力オブジェクト。これらの厳格なルールは SubscriptionFilter 式オブジェクトに含まれるため、少なくとも 1 つのルールが満たされてサブスクリプションフィルターを通過することになります。
>
> レスポンス
> 戻り値は SubscriptionFilter。

util.transform.toSubscriptionFilter(Map, List)

Map 入力オブジェクトを SubscriptionFilter 式オブジェクトに変換します。util.transform.toSubscriptionFilter メソッドはextensions.setSubscriptionFilter() 拡張子への入力として使用されます。詳細については、「拡張子の使用」を参照してください。

1 番目の引数は、SubscriptionFilter 式オブジェクトに変換される Map 入力オブジェクトです。2 番目の引数は、SubscriptionFilter 式オブジェクトを作成する際に 1 番目の Map 入力オブジェクトでは無視されるフィールド名の List です。

util.transform.toSubscriptionFilter(Map, List, Map)

Map 入力オブジェクトを SubscriptionFilter 式オブジェクトに変換します。util.transform.toSubscriptionFilter メソッドはextensions.setSubscriptionFilter() 拡張子への入力として使用されます。詳細については、「拡張子の使用」を参照してください。

util.transform.toDynamoDBConditionExpression(conditionObject)

DynamoDB の条件式を作成します。

サブスクリプションフィルター引数

以下の表では、以下のユーティリティの引数の定義方法について説明しています。

- Util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter

Argument 1: Map

引数 1 は、以下のキー値を持つ Map オブジェクトです。

- フィールド名
- "and"
- "or"

フィールド名をキーとする場合、これらのフィールドのエントリの条件は "operator" : "value" という形式になります。

次の例では、Map にエントリを追加する方法を示します。

```
"field_name" : {
                    "operator1" : value
                }

## We can have multiple conditions for the same field_name:

"field_name" : {
                    "operator1" : value
                    "operator2" : value
```

```
                    .
                    .
                    .
                }
```

フィールドに 2 つ以上の条件が設定されている場合、これらの条件はすべて OR 演算を使用するものとみなされます。

入力 Map には「and」と「or」をキーとして使用することもできます。つまり、その中のすべてのエントリは、キーに応じて AND または OR ロジックを使用して結合する必要があります。キー値「and」と「or」には条件の配列が必要です。

```
"and" : [

            {
                "field_name1" : {
                    "operator1" : value
                }
             },

             {
                "field_name2" : {
                    "operator1" : value
                }
             },
             .
             .
        ].
```

「and」と「or」はネストできることに注意してください。つまり、「and/or」を別の「and/or」ブロック内にネストしてもかまいません。ただし、これは単純なフィールドでは機能しません。

```
"and" : [

            {
                "field_name1" : {
                    "operator" : value
                }
             },

             {
                "or" : [
```

```
                            {
                                "field_name2" : {
                                    "operator" : value
                                }
                            },

                            {
                                "field_name3" : {
                                    "operator" : value
                                }
                            }

                        ].
```

次の例は、util.transform.toSubscriptionFilter(Map) : Mapを使用して引数 1 を入力したものです。

入力

引数 1: マップ:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
```

```
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

出力

結果は Map オブジェクトです。

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Admin"
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
```

```
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "gte",
          "value": 20
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Admin"
        }
      ]
    },
    {
```

```
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "gte",
          "value": 20
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 2: List

引数 2 には、SubscriptionFilter 式オブジェクトを作成するときに入力 Map (引数 1) で考慮してはいけないフィールド名の List が含まれています。List は空になることもあります。

次の例は、util.transform.toSubscriptionFilter(Map, List) : Map を使用した引数 1 と引数 2 の入力を示しています。

入力

引数 1: マップ:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
```

```
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

引数 2: リスト:

```
["percentageUp", "author"]
```

出力

結果は Map オブジェクトです。

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
```

```
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 3: Map

引数 3 は、フィールド名をキー値とする Map オブジェクトです (「and」や「or」は使用できません)。フィールド名がキーの場合、これらのフィールドの条件は "operator" : "value" という形式のエントリになります。引数 1 とは異なり、引数 3 では同じキーに複数の条件を設定できません。さらに、引数 3 には「and」や「or」句がないため、ネストも必要ありません。

引数 3 は厳密な規則のリストを表し、これらの条件の少なくとも 1 つが満たされてフィルタを通過するように SubscriptionFilter 式オブジェクトに追加されます。

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.
```

次の例は、util.transform.toSubscriptionFilter(Map, List, Map) : Map を使用した引数 1、引数 2、引数 3 の入力を示しています。

入力

引数 1: マップ:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

引数 2: リスト:

```
["percentageUp", "author"]
```

引数 3: マップ:

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
```

```
    "eq": "Person1"
  }
}
```

出力

結果は Map オブジェクトです。

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        },
        {
          "fieldName": "upvotes",
          "operator": "gte",
          "value": 250
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
```

```
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Person1"
        }
      ]
    }
  ]
}
```

util.str の文字列ヘルパー

`util.str` には一般的な文字列操作を支援するメソッドが含まれています。

util.str utils リスト

`util.str.normalize(String, String)`

NFC、NFD、NFKC、または NFKD の 4 つのユニコード正規化形式のいずれかを使用して文字列を正規化します。最初の引数は正規化する文字列です。2 番目の引数は、正規化プロセスに使用する正規化タイプを指定する「nfc」「nfd」「nfkc」「nfkd」のいずれかです。

拡張子

`extensions` には、リゾルバー内で追加のアクションを行うためのメソッドセットが含まれています。

拡張子の使用

`extensions.evictFromApiCache(typeName: string, fieldName: string, keyValuePair: Record<string, string>) : Object`

AWS AppSync サーバー側のキャッシュから項目を削除します。最初の引数は型名です。2 番目の引数はフィールド名です。3 番目の引数は、キャッシュキー値を指定するキーと値のペ

ア項目を含むオブジェクトです。オブジェクト内の項目は、キャッシュされたリゾルバーの cachingKey のキャッシュキーと同じ順序で配置する必要があります。キャッシュの詳細については、「キャッシング動作」を参照してください。

例 1:

この例では、呼び出されたキャッシュキー context.arguments.semester が使用されたリゾルバー Query.allClasses に対してキャッシュされた項目がエビクションされています。ミューテーションが呼び出されてリゾルバーが実行され、エントリが正常にクリアされると、レスポンスには拡張子オブジェクトに、削除されたエントリの数を示す apiCacheEntriesDeleted 値が含まれます。

```
import { util, extensions } from '@aws-appsync/utils';

export const request = (ctx) => ({ payload: null });

export function response(ctx) {
 extensions.evictFromApiCache('Query', 'allClasses', {
  'context.arguments.semester': ctx.args.semester,
 });
 return null;
}
```

> **Note**
>
> この関数はミューテーションに対してのみ機能し、クエリでは機能しません。

サブスクリプションの延長

extensions.setSubscriptionFilter(filterJsonObject)

強化されたサブスクリプションフィルターを定義します。各サブスクリプション通知イベントは、提供されたサブスクリプションフィルタに対して評価され、すべてのフィルタが true に評価された場合、クライアントに通知を配信します。引数は filterJsonObject です (この引数の詳細については、以下の「引数: filterJsonObject」セクションを参照してください)。「強化されたサブスクリプションのフィルタリング」を参照してください。

> **Note**
>
> この拡張関数は、サブスクリプションリゾルバーのレスポンスハンドラーでのみ使用できます。また、フィルターの作成にも `util.transform.toSubscriptionFilter` を使用することをおすすめします。

`extensions.setSubscriptionInvalidationFilter(filterJsonObject)`

サブスクリプション無効化フィルターを定義します。サブスクリプションフィルタは無効化ペイロードに照らして評価されてから、フィルターが `true` と評価された場合、与えられたサブスクリプションを無効にします。引数は `filterJsonObject` です (この引数の詳細については、以下の「引数: filterJsonObject」セクションを参照してください)。「強化されたサブスクリプションのフィルタリング」を参照してください。

> **Note**
>
> この拡張関数は、サブスクリプションリゾルバーのレスポンスハンドラーでのみ使用できます。フィルターの作成にも `util.transform.toSubscriptionFilter` を使用することをおすすめします。

`extensions.invalidateSubscriptions(invalidationJsonObject)`

ミューテーションによるサブスクリプションの無効化を開始するのに使用します。引数は `invalidationJsonObject` です (この引数の詳細については、以下の「引数: InvalidationJsonObject」セクションを参照してください)。

> **Note**
>
> この拡張関数はミューテーションリゾルバーのレスポンスマッピングテンプレートでのみ使用できます。
> 1 つのリクエストで使用できるユニークな `extensions.invalidateSubscriptions()` メソッド呼び出しは 5 つまでです。この制限を超えた場合、GraphQL エラーが表示されます。

引数: filterJsonObject

JSON オブジェクトは、サブスクリプションフィルターまたは無効化フィルターのいずれかを定義します。これは、filterGroup 内のフィルターの配列です。各フィルターは個別のフィルターの集まりです。

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

各フィルターには次の 3 つの属性があります。

- fieldName – GraphQL スキーマフィールド。
- operator – オペレータータイプ。
- value – サブスクリプション通知 fieldName 値と比較する値。

以下は、これらの属性の割り当ての例です。

```
{
 "fieldName" : "severity",
```

```
 "operator" : "le",
 "value" : context.result.severity
}
```

引数: invalidationJsonObject

`invalidationJsonObject` では以下が定義されます。

- `subscriptionField` – 無効にする GraphQL スキーマのサブスクリプション。`subscriptionField`で文字列として定義されている 1 つのサブスクリプションは無効とみなされます。
- `payload` — 無効化フィルターがその値に対して `true` と評価された場合に、サブスクリプションを無効化するための入力として使用されるキーと値のペアリスト。

  以下の例では、サブスクリプションリゾルバーで定義された無効化フィルターが `payload` 値に対して `true` と評価されたとき、`onUserDelete` サブスクリプションを使用してサブスクライブして接続しているクライアントを無効にします。

```
export const request = (ctx) => ({ payload: null });

export function response(ctx) {
 extensions.invalidateSubscriptions({
  subscriptionField: 'onUserDelete',
  payload: { group: 'Developer', type: 'Full-Time' },
 });
 return ctx.result;
}
```

util.xml の XML ヘルパー

`util.xml`には、XML 文字列変換に役立つメソッドが含まれています。

util.xml utils リスト

`util.xml.toMap(String) : Object`

XML 文字列を辞書に変換します。

例 1:

```
Input:

<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
    <id>1</id>
    <title>Getting started with GraphQL</title>
</post>
</posts>

Output (object):

{
    "posts":{
      "post":{
        "id":1,
        "title":"Getting started with GraphQL"
      }
    }
}
```

例 2:

```
Input:

<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
<post>
  <id>2</id>
  <title>Getting started with AppSync</title>
</post>
</posts>

Output (JavaScript object):

{
    "posts":{
    "post":[
        {
```

```
            "id":1,
            "title":"Getting started with GraphQL"
        },
        {
            "id":2,
            "title":"Getting started with AppSync"
        }
    ]
    }
}
```

util.xml.toJsonString(String, Boolean?) : String

XML 文字列を JSON 文字列に変換します。これは、出力が文字列である点を除き、toMap に似ています。これは、XML レスポンスを HTTP オブジェクトから JSON に直接変換し、返す場合に便利です。オプションの boolean パラメータを設定して、JSON を文字列でエンコードするかどうかを決定できます。

DynamoDB のAWS AppSync JavaScript リゾルバー関数リファレンス

AWS AppSync DynamoDB 関数を使用すると、受信 GraphQL リクエストを DynamoDB 呼び出しにマッピングし、DynamoDB レスポンスを GraphQL にマッピングすることで、GraphQL を使用してアカウント内の既存の Amazon DynamoDB テーブルにデータを保存および取得できます。このセクションでは、サポートされる DynamoDB オペレーションのリクエストハンドラーとレスポンスハンドラーについて説明します。

- GetItem - GetItem リクエストでは、DynamoDB 関数に DynamoDB への GetItem リクエストを行うように指示し、DynamoDB 内の項目のキーと、一貫した読み取りを使用するかどうかを指定できます。
- PutItem - PutItem リクエストマッピングドキュメントでは、DynamoDB 関数に DynamoDB への PutItem リクエストを行うように指示し、DynamoDB の項目のキー、項目の全内容 (キーと attributeValues で構成される)、およびオペレーションが成功するための条件を指定できます。
- UpdateItem - UpdateItem リクエストでは、DynamoDB 関数に DynamoDB への UpdateItem リクエストを行うように指示し、DynamoDB の項目のキー、DynamoDB の項目を更新する方法を説明する更新式、およびオペレーションが成功するための条件を指定できます。

- DeleteItem - DeleteItem リクエストでは、DynamoDB 関数に DynamoDB に DeleteItem リクエストを行うように指示し、DynamoDB 内の項目のキーと、オペレーションが成功するための条件を指定できます。
- Query - Query リクエストオブジェクトを使用すると、DynamoDB リゾルバーに DynamoDB へのクエリリクエストを行うように指示できます。また、キー式、使用するインデックス、追加のフィルター、返す項目の数、一貫した読み取りを使用するかどうか、クエリの方向 (前方または後方)、ページ分割トークンを指定できます。
- Scan - Scan リクエストは、DynamoDB 関数に DynamoDB への Scan リクエストを指示することができ、結果を除外するフィルタ、使用するインデックス、返すアイテムの数、一貫性のある読み取りを使用するかどうか、ページ分割トークン、並列スキャンを指定することができます。
- Sync - Sync リクエストオブジェクトを使用すると、DynamoDB テーブルからすべての結果を取得し、最後のクエリ (差分更新) 以降に変更されたデータのみを受け取ることができます。Sync リクエストは、バージョン管理された DynamoDB データソースに対してのみ行うことができます。フィルタを指定して、結果、返す項目の数、ページ分割トークン、および最後の Sync オペレーションが開始された日時を除外できます。
- BatchGetItem - BatchGetItem リクエストオブジェクトを使用すると、DynamoDB の関数から DynamoDB への BatchGetItem リクエストで、複数のテーブルから複数の項目を取得するように指定できます。このリクエストオブジェクトでは、項目を取得するテーブル名と、各テーブルから取得する項目のキーを指定する必要があります。
- BatchDeleteItem - BatchDeleteItem リクエストオブジェクトを使用すると、DynamoDB の関数から DynamoDB への BatchWriteItem リクエストで、複数のテーブルから複数の項目を削除するように指定できます。このリクエストオブジェクトでは、項目を削除するテーブル名と、各テーブルから削除する項目のキーを指定する必要があります。
- BatchPutItem - BatchPutItem リクエストオブジェクトを使用すると、DynamoDB の関数から DynamoDB への BatchWriteItem リクエストオブジェクトを使用すると リクエストで、複数のテーブルに複数の項目を配置するように指定できます。このリクエストオブジェクトでは、項目を配置するテーブル名と、各テーブルに配置する完全な項目を指定する必要があります。
- TransactGetItems - TransactGetItems リクエストオブジェクトを使用すると、DynamoDB の関数から DynamoDB への TransactGetItems リクエストで、複数のテーブルから複数の項目を取得するように指定できます。このリクエストオブジェクトでは、項目を取得する各リクエスト項目のテーブル名と、各テーブルから取得する各リクエスト項目のキーを指定する必要があります。
- TransactWriteItems - TransactWriteItems リクエストオブジェクトを使用すると、DynamoDB の関数から DynamoDB への TransactWriteItems リクエストで、複数のテーブルに複数の項目を書き込むように指定できます。このリクエストオブジェクトでは、各リクエスト項目の宛先テーブル名、

実行する各リクエスト項目のオペレーション、および書き込む各リクエスト項目のキーを指定する必要があります。

- 型システム (リクエストマッピング) - DynamoDB 型付けが AWS AppSync リクエストにどのように統合されるかについて詳しく説明します。
- 型システム (レスポンスマッピング) - DynamoDB タイプがレスポンスペイロードで GraphQL または JSON に自動的に変換される方法について詳しく説明します。
- フィルター - クエリおよびスキャンオペレーションのフィルターの詳細について説明します。
- 条件式 - PutItem オペレーション、UpdateItem オペレーション、DeleteItem オペレーションの条件式について説明します。
- トランザクション条件式 - TransactWriteItems オペレーションの条件式について説明します。
- プロジェクション - 読み取りオペレーションで属性を指定する方法について説明します。

GetItem

`GetItem` リクエストでは、 AWS AppSync DynamoDB 関数に DynamoDB への`GetItem`リクエストを行うように指示し、以下を指定できます。

- DynamoDB の項目のキー
- 整合性のある読み込みを使用するかどうか

`GetItem` リクエストの構造は次のとおりです。

```
type DynamoDBGetItem = {
  operation: 'GetItem';
  key: { [key: string]: any };
  consistentRead?: ConsistentRead;
  projection?: {
    expression: string;
    expressionNames?: { [key: string]: string };
  };
};
```

各フィールドの定義は以下のようになります。

GetItem フィールド

GetItem フィールドリスト

`operation`

実行する DynamoDB の処理。`GetItem` DynamoDB の処理を実行するには、これを `GetItem` に設定する必要があります。この値は必須です。

`key`

DynamoDB の項目のキー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

`consistentRead`

DynamoDB で強力な整合性のある読み込みを実行するかどうかを示します。これはオプションであり、デフォルトは `false` です。

`projection`

DynamoDB オペレーションから返される属性を指定するために使用されるプロジェクション。プロジェクションの詳細については、「プロジェクション」を参照してください。このフィールドはオプションです。

DynamoDB から返された項目が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、コンテキス結果ト (`context.result`) で参照できます。

DynamoDB の型変換の詳細については、「型システム (レスポンスマッピング)」を参照してください。

JavaScript リゾルバーの詳細については、「JavaScript リゾルバーの概要」を参照してください。

例

次の例は、GraphQL `getThing(foo: String!, bar: String!)` クエリの関数リクエストハンドラーです。

```
export function request(ctx) {
  const {foo, bar} = ctx.args
  return {
```

```
    operation : "GetItem",
    key : util.dynamodb.toMapValues({foo, bar}),
    consistentRead : true
  }
}
```

DynamoDB GetItem API の詳細については、「DynamoDB API のドキュメント」を参照してください。

PutItem

PutItem リクエストマッピングドキュメントでは、 AWS AppSync DynamoDB 関数に DynamoDB へのPutItemリクエストを行うように指示し、以下を指定できます。

- DynamoDB の項目のキー
- 項目の完全なコンテンツ (key および attributeValues で構成されます)
- 処理が成功する条件

PutItem リクエストの構造は次のとおりです。

```
type DynamoDBPutItemRequest = {
  operation: 'PutItem';
  key: { [key: string]: any };
  attributeValues: { [key: string]: any};
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

各フィールドの定義は以下のようになります。

PutItem フィールド

PutItem フィールドリスト

operation

実行する DynamoDB の処理。PutItem DynamoDB の処理を実行するには、これを PutItem に設定する必要があります。この値は必須です。

key

DynamoDB の項目のキー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

attributeValues

DynamoDB に渡す項目の残りの属性です。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。このフィールドはオプションです。

condition

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件が指定されていない場合、PutItem リクエストはその項目の既存のエントリを上書きします。条件の詳細については、「条件式」を参照してください。この値はオプションです。

_version

項目の既知の最新バージョンを表す数値。この値はオプションです。このフィールドは競合の検出に使用され、バージョン管理されたデータソースでのみサポートされます。

customPartitionKey

有効にすると、この文字列値は、バージョニングが有効になっているときにデルタ同期テーブルで使用される ds_sk および ds_pk レコードの形式を変更します (詳細については、「AWS AppSyncデベロッパーガイド」の「競合検出と同期」を参照)。有効にすると、populateIndexFields エントリの処理も有効になります。このフィールドはオプションです。

populateIndexFields

ブール値で、**customPartitionKey** と一緒に有効にすると、差分同期テーブル、具体的には gsi_ds_pk と gsi_ds_sk 列のレコードごとに新しいエントリが作成されます。詳細については、「AWS AppSyncデベロッパーガイド」の「競合検出と同期」を参照してください。このフィールドはオプションです。

DynamoDB に書き込まれた項目が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、コンテキスト結果 (context.result) で参照できます。

DynamoDB に書き込まれた項目が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、コンテキスト結果 (`context.result`) で参照できます。

DynamoDB の型変換の詳細については、「型システム (レスポンスマッピング)」を参照してください。

JavaScript リゾルバーの詳細については、「JavaScript リゾルバーの概要」を参照してください。

例 1

次の例は、GraphQL ミューテーション `updateThing(foo: String!, bar: String!, name: String!, version: Int!)` の関数リクエストハンドラーです。

指定したキーに対応する項目がない場合は、作成されます。指定したキーに対応する項目がすでにある場合は、上書きされます。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, ...values} = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}
```

例 2

次の例は、GraphQL ミューテーション `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)` の関数リクエストハンドラーです。

この例では、DynamoDB に現在ある項目の `version` フィールドに `expectedVersion` が設定されていることを確認します。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, name, expectedVersion } = ctx.args;
  const values = { name, version: expectedVersion + 1 };
  let condition = util.transform.toDynamoDBConditionExpression({
    version: { eq: expectedVersion },
```

```
  });

  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ foo, bar }),
    attributeValues: util.dynamodb.toMapValues(values),
    condition,
  };
}
```

DynamoDB PutItem API の詳細については、「DynamoDB API のドキュメント」を参照してください。

UpdateItem

UpdateItem リクエストにより、 AWS AppSync DynamoDB 関数に DynamoDB へのUpdateItemリクエストを行うように指示し、以下を指定できます。

- DynamoDB の項目のキー
- DynamoDB の項目を更新する方法を示す更新式
- 処理が成功する条件

UpdateItem リクエストの構造は次のとおりです。

```
type DynamoDBUpdateItemRequest = {
  operation: 'UpdateItem';
  key: { [key: string]: any };
  update: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

各フィールドの定義は以下のようになります。

UpdateItem フィールド

UpdateItem フィールドリスト

`operation`

実行する DynamoDB の処理。`UpdateItem` DynamoDB の処理を実行するには、これを `UpdateItem` に設定する必要があります。この値は必須です。

`key`

DynamoDB の項目のキー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

`update`

`update` セクションには、 DynamoDBの項目の更新方法を示す更新式を指定することができます。更新式の記述方法の詳細については、「DynamoDB UpdateExpressions」のドキュメントを参照してください。このセクションは必須です。

`update` セクションには次の 3 つのコンポーネントがあります。

`expression`

更新式です。この値は必須です。

`expressionNames`

式の属性名のプレースホルダーを示します。キー - 値のペアの形式になります。キーは `expression` で使用される名前のプレースホルダーに対応し、値は DynamoDB の項目の属性名と一致する文字列でなければなりません。このフィールドはオプションであり、`expression` で使用される式の属性名のプレースホルダーのみを入力します。

`expressionValues`

式の属性値のプレースホルダーを示します。キー - 値のペアの形式になります。キーは `expression` で使用される値のプレースホルダーに対応し、値は型付き値でなければなりません。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この指定は必須です。このフィールドはオプションであり、`expression` で使用される式の属性値のプレースホルダーのみを入力します。

condition

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件を指定していない場合は、UpdateItem リクエストによって、現在の状態にかかわらず、既存のエントリが更新されます。条件の詳細については、「条件式」を参照してください。この値はオプションです。

_version

項目の既知の最新バージョンを表す数値。この値はオプションです。このフィールドは競合の検出に使用され、バージョン管理されたデータソースでのみサポートされます。

customPartitionKey

有効にすると、この文字列値は、バージョニングが有効になっているときにデルタ同期テーブルで使用される ds_sk および ds_pk レコードの形式を変更します (詳細については、「AWS AppSyncデベロッパーガイド」の「競合検出と同期」を参照)。有効にすると、populateIndexFields エントリの処理も有効になります。このフィールドはオプションです。

populateIndexFields

ブール値で、**customPartitionKey** と一緒に有効にすると、差分同期テーブル、具体的には gsi_ds_pk と gsi_ds_sk 列のレコードごとに新しいエントリが作成されます。詳細については、「AWS AppSyncデベロッパーガイド」の「競合検出と同期」を参照してください。このフィールドはオプションです。

DynamoDB の更新された項目が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、コンテキスト結果 (context.result) で参照できます。

DynamoDB の型変換の詳細については、「型システム (リクエストマッピング)」を参照してください。

JavaScript リゾルバーの詳細については、「JavaScript リゾルバーの概要」を参照してください。

例 1

次の例は、GraphQL ミューテーション upvote(id: ID!) の関数リクエストハンドラーです。

この例では、DynamoDB の項目の upvotes フィールドと version フィールドが 1 ずつ増加されます。

```
import { util } from '@aws-appsync/utils';
```

```
export function request(ctx) {
  const { id } = ctx.args;
  return {
    operation: 'UpdateItem',
    key: util.dynamodb.toMapValues({ id }),
    update: {
      expression: 'ADD #votefield :plusOne, version :plusOne',
      expressionNames: { '#votefield': 'upvotes' },
      expressionValues: { ':plusOne': { N: 1 } },
    },
  };
}
```

例 2

次の例は、GraphQL ミューテーション `updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)` の関数リクエストハンドラーです。

これは、引数を確認して、クライアントから入力された引数のみを含む更新式を動的に生成する複雑な例です。たとえば、`title` と `author` を省略すると、それらは更新されません。引数が指定されているが、その値が `null` の場合、そのフィールドは DynamoDB のオブジェクトから削除されます。最後に、この処理内の条件によって、DynamoDB に現在ある項目の `version` フィールドに `expectedVersion` が設定されているかどうかを確認します。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { args: { input: { id, ...values } } } = ctx;

  const condition = {
    id: { attributeExists: true },
    version: { eq: values.expectedVersion },
  };
  values.expectedVersion += 1;
  return dynamodbUpdateRequest({ keys: { id }, values, condition });
}


/**
 * Helper function to update an item
 * @returns an UpdateItem request
 */
function dynamodbUpdateRequest(params) {
```

```
  const { keys, values, condition: inCondObj } = params;

  const sets = [];
  const removes = [];
  const expressionNames = {};
  const expValues = {};

  // Iterate through the keys of the values
  for (const [key, value] of Object.entries(values)) {
    expressionNames[`#${key}`] = key;
    if (value) {
      sets.push(`#${key} = :${key}`);
      expValues[`:${key}`] = value;
    } else {
      removes.push(`#${key}`);
    }
  }

  let expression = sets.length ? `SET ${sets.join(', ')}` : '';
  expression += removes.length ? ` REMOVE ${removes.join(', ')}` : '';

  const condition = JSON.parse(
    util.transform.toDynamoDBConditionExpression(inCondObj)
  );

  return {
    operation: 'UpdateItem',
    key: util.dynamodb.toMapValues(keys),
    condition,
    update: {
      expression,
      expressionNames,
      expressionValues: util.dynamodb.toMapValues(expValues),
    },
  };
}
```

DynamoDB UpdateItem API の詳細については、「DynamoDB API のドキュメント」を参照してください。

DeleteItem

DeleteItem リクエストでは、 AWS AppSync DynamoDB 関数に DynamoDB へのDeleteItemリクエストを行うように指示し、以下を指定できます。

- DynamoDB の項目のキー
- 処理が成功する条件

DeleteItem リクエストの構造は次のとおりです。

```
type DynamoDBDeleteItemRequest = {
  operation: 'DeleteItem';
  key: { [key: string]: any };
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

各フィールドの定義は以下のようになります。

DeleteItem フィールド

DeleteItem フィールドリスト

operation

実行する DynamoDB の処理。DeleteItem DynamoDB の処理を実行するには、これを DeleteItem に設定する必要があります。この値は必須です。

key

DynamoDB の項目のキー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

condition

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件を指定していない場合、DeleteItem リクエストによって、現在の状態に

かかわらず、項目が削除されます。条件の詳細については、「条件式」を参照してください。この値はオプションです。

_version

項目の既知の最新バージョンを表す数値。この値はオプションです。このフィールドは競合の検出に使用され、バージョン管理されたデータソースでのみサポートされます。

customPartitionKey

有効にすると、この文字列値は、バージョニングが有効になっているときにデルタ同期テーブルで使用される ds_sk および ds_pk レコードの形式を変更します (詳細については、「AWS AppSyncデベロッパーガイド」の「競合検出と同期」を参照)。有効にすると、populateIndexFields エントリの処理も有効になります。このフィールドはオプションです。

populateIndexFields

ブール値で、**customPartitionKey** と一緒に有効にすると、差分同期テーブル、具体的には gsi_ds_pk と gsi_ds_sk 列のレコードごとに新しいエントリが作成されます。詳細については、「AWS AppSyncデベロッパーガイド」の「競合検出と同期」を参照してください。このフィールドはオプションです。

DynamoDB から削除された項目が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、コンテキスト結果 (context.result) で参照できます。

DynamoDB の型変換の詳細については、「型システム (リクエストマッピング)」を参照してください。

JavaScript リゾルバーの詳細については、「JavaScript リゾルバーの概要」を参照してください。

例 1

次の例は、GraphQL ミューテーション deleteItem(id: ID!) の関数リクエストハンドラーです。この ID に対応する項目がある場合は、削除されます。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'DeleteItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
```

```
}
```

例 2

次の例は、GraphQL ミューテーション `deleteItem(id: ID!, expectedVersion: Int!)` の関数リクエストハンドラーです。この ID に対応する項目がある場合は、その `version` フィールドに `expectedVersion` が設定されているときにのみ削除されます。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { id, expectedVersion } = ctx.args;
  const condition = {
    id: { attributeExists: true },
    version: { eq: expectedVersion },
  };
  return {
    operation: 'DeleteItem',
    key: util.dynamodb.toMapValues({ id }),
    condition: util.transform.toDynamoDBConditionExpression(condition),
  };
}
```

DynamoDB `DeleteItem` API の詳細については、「DynamoDB API のドキュメント」を参照してください。

クエリ

`Query` リクエストオブジェクトを使用すると、 AWS AppSync DynamoDB リゾルバーに DynamoDB への`Query`リクエストを行うように指示し、以下を指定できます。

- キー式
- 使用するインデックス
- 任意の追加フィルタ
- 返す項目の数
- 整合性のある読み込みを使用するかどうか
- クエリの方向 (前方または後方)
- ページ分割トークン

Query リクエストオブジェクトのノードの構造は次のとおりです。

```
type DynamoDBQueryRequest = {
  operation: 'Query';
  query: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
  index?: string;
  nextToken?: string;
  limit?: number;
  scanIndexForward?: boolean;
  consistentRead?: boolean;
  select?: 'ALL_ATTRIBUTES' | 'ALL_PROJECTED_ATTRIBUTES' | 'SPECIFIC_ATTRIBUTES';
  filter?: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
  projection?: {
    expression: string;
    expressionNames?: { [key: string]: string };
  };
};
```

各フィールドの定義は以下のようになります。

Query フィールド

Query フィールドリスト

`operation`

実行する DynamoDB の処理。`Query` DynamoDB の処理を実行するには、これを `Query` に設定する必要があります。この値は必須です。

`query`

`query` セクションには、DynamoDB から取得する項目を指示するキー条件式を指定することができます。キー条件式の記述方法の詳細については、DynamoDB KeyConditions のドキュメントを参照してください。このセクションの指定は必須です。

expression

クエリ式です。このフィールドの指定は必須です。

expressionNames

式の属性名のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される名前のプレースホルダーに対応し、値は DynamoDB の項目の属性名と一致する文字列でなければなりません。このフィールドはオプションであり、expression で使用される式の属性名のプレースホルダーのみを入力します。

expressionValues

式の属性値のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される値のプレースホルダーに対応し、値は型付き値でなければなりません。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。このフィールドはオプションであり、expression で使用される式の属性値のプレースホルダーのみを入力します。

filter

DynamoDB からの結果が返される前に、その結果をフィルタリングするために使用する追加フィルタです。フィルタの詳細については、「フィルタ」を参照してください。このフィールドはオプションです。

index

クエリを実行するインデックスの名前です。DynamoDB クエリの処理により、ハッシュキーのプライマリキーインデックスに加えて、ローカルセカンダリインデックスとグローバルセカンダリインデックスをスキャンできます。指定されると、DynamoDB が指定されたインデックスにクエリを実行します。省略すると、プライマリキーインデックスに対してクエリが実行されます。

nextToken

前のクエリを継続するためのページ分割トークンです。これは前のクエリから取得されます。このフィールドはオプションです。

limit

評価する項目の最大数 (一致する項目の数であるとは限りません)。このフィールドはオプションです。

scanIndexForward

クエリを前方と後方のどちらに実行するかを示すブール値です。このフィールドはオプションであり、デフォルトは true です。

consistentRead

DynamoDB にクエリを実行する際に整合性のある読み込みを使用するかどうかを示すブール値です。このフィールドはオプションであり、デフォルトは `false` です。

select

デフォルトでは、 AWS AppSync DynamoDB リゾルバーはインデックスに射影された属性のみを返します。より多くの属性が必要な場合に、このフィールドを設定できます。このフィールドはオプションです。サポートされている値には以下があります。

ALL_ATTRIBUTES

指定されたテーブルまたはインデックスのすべての項目の属性を返します。ローカルセカンダリインデックスに対してクエリを実行する場合、DynamoDB は、親のテーブルからインデックスの項目に一致したすべての項目をフェッチします。インデックスがすべての項目の属性を射影するように設定されている場合、すべてのデータはローカルセカンダリインデックスから取得されるため、フェッチは必要ありません。

ALL_PROJECTED_ATTRIBUTES

インデックスにクエリを実行する場合のみ使用できます。インデックスに投射されたすべての属性を取得します。インデックスがすべての属性を投射するように設定されている場合、この返り値は `ALL_ATTRIBUTES` を指定した場合と同等になります。

SPECIFIC_ATTRIBUTES

`projection` の `expression` にリストされている属性のみを返します。この戻り値は、`Select` の値を指定せずに `projection` の `expression` を指定するのと同じです。

projection

DynamoDB オペレーションから返される属性を指定するために使用されるプロジェクション。プロジェクションの詳細については、「プロジェクション」を参照してください。このフィールドはオプションです。

DynamoDB からの結果が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、コンテキスト結果 (`context.result`) で参照できます。

DynamoDB の型変換の詳細については、「型システム (リクエストマッピング)」を参照してください。

JavaScript リゾルバーの詳細については、「JavaScript リゾルバーの概要」を参照してください。

結果は以下の構造を持ちます。

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

各フィールドの定義は以下のようになります。

items

DynamoDB クエリで返された項目を含むリストです。

nextToken

さらに結果がある場合、nextToken には別のリクエストで使用できるページ分割トークンが含まれています。 AWS AppSync は、DynamoDB から返されるページ分割トークンを暗号化して難読化します。これにより、テーブルデータが誤って呼び出し元に漏えいされるのを防ぎます。また、これらのページ分割トークンは、異なる関数やリゾルバー間では使用できないことにも注意してください。

scannedCount

フィルタ式 (ある場合) が適用される前に、クエリの条件式に一致した項目の数です。

例

次の例は、GraphQL クエリgetPosts(owner: ID!) の関数リクエストハンドラーです。

この例では、テーブルのグローバルセカンダリインデックスにクエリが実行され、指定した ID が所有するすべての投稿が返されます。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { owner } = ctx.args;
  return {
    operation: 'Query',
    query: {
      expression: 'ownerId = :ownerId',
      expressionValues: util.dynamodb.toMapValues({ ':ownerId': owner }),
```

```
    },
    index: 'owner-index',
  };
}
```

DynamoDB Query API の詳細については、「DynamoDB API のドキュメント」を参照してください。

Scan

Scan リクエストでは、 AWS AppSync DynamoDB 関数に DynamoDB へのScanリクエストを行うように指示し、以下を指定できます。

- 結果を除外するフィルタ
- 使用するインデックス
- 返す項目の数
- 整合性のある読み込みを使用するかどうか
- ページ分割トークン
- 並列スキャン

Scan リクエストオブジェクトのノードの構造は次のとおりです。

```
type DynamoDBScanRequest = {
  operation: 'Scan';
  index?: string;
  limit?: number;
  consistentRead?: boolean;
  nextToken?: string;
  totalSegments?: number;
  segment?: number;
  filter?: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
  projection?: {
    expression: string;
    expressionNames?: { [key: string]: string };
  };
```

```
};
```

各フィールドの定義は以下のようになります。

Scan フィールド

Scan フィールドリスト

`operation`

実行する DynamoDB の処理。`Scan` DynamoDB の処理を実行するには、これを `Scan` に設定する必要があります。この値は必須です。

`filter`

DynamoDB からの結果が返される前に、その結果をフィルタリングするために使用するフィルタです。フィルタの詳細については、「フィルタ」を参照してください。このフィールドはオプションです。

`index`

クエリを実行するインデックスの名前です。DynamoDB クエリの処理により、ハッシュキーのプライマリキーインデックスに加えて、ローカルセカンダリインデックスとグローバルセカンダリインデックスをスキャンできます。指定されると、DynamoDB が指定されたインデックスにクエリを実行します。省略すると、プライマリキーインデックスに対してクエリが実行されます。

`limit`

一度に評価する項目の最大数です。このフィールドはオプションです。

`consistentRead`

DynamoDB にクエリを実行する際に整合性のある読み込みを使用するかどうかを示すブール値です。このフィールドはオプションであり、デフォルトは `false` です。

`nextToken`

前のクエリを継続するためのページ分割トークンです。これは前のクエリから取得されます。このフィールドはオプションです。

`select`

デフォルトでは、 AWS AppSync DynamoDB 関数はインデックスに射影される属性のみを返します。より多くの属性が必要な場合にこのフィールドを設定します。このフィールドはオプションです。サポートされている値には以下があります。

ALL_ATTRIBUTES

指定されたテーブルまたはインデックスのすべての項目の属性を返します。ローカルセカンダリインデックスに対してクエリを実行する場合、DynamoDB は、親のテーブルからインデックスの項目に一致したすべての項目をフェッチします。インデックスがすべての項目の属性を射影するように設定されている場合、すべてのデータはローカルセカンダリインデックスから取得されるため、フェッチは必要ありません。

ALL_PROJECTED_ATTRIBUTES

インデックスにクエリを実行する場合のみ使用できます。インデックスに投射されたすべての属性を取得します。インデックスがすべての属性を投射するように設定されている場合、この返り値は `ALL_ATTRIBUTES` を指定した場合と同等になります。

SPECIFIC_ATTRIBUTES

`projection` の `expression` にリストされている属性のみを返します。この戻り値は、`Select` の値を指定せずに `projection` の `expression` を指定するのと同じです。

totalSegments

並列スキャンが実行されるまでにテーブルを分割するセグメントの数です。このフィールドはオプションですが、`segment` を指定した場合には指定する必要があります。

segment

並列スキャンの実行時のこの処理でのテーブルセグメントです。このフィールドはオプションですが、`totalSegments` を指定した場合には指定する必要があります。

projection

DynamoDB オペレーションから返される属性を指定するために使用されるプロジェクション。プロジェクションの詳細については、「プロジェクション」を参照してください。このフィールドはオプションです。

DynamoDB スキャンにより返された結果が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、コンテキスト結果 (`context.result`) で参照できます。

DynamoDB の型変換の詳細については、「型システム (リクエストマッピング)」を参照してください。

JavaScript リゾルバーの詳細については、「JavaScript リゾルバーの概要」を参照してください。

結果は以下の構造を持ちます。

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

各フィールドの定義は以下のようになります。

items

DynamoDB スキャンにより返された項目を含むリストです。

nextToken

さらに結果がある場合、 には別のリクエストで使用できるページ分割トークンnextTokenが含まれています。 AWS AppSync は、DynamoDB から返されたページ分割トークンを暗号化して難読化します。これにより、テーブルデータが誤って呼び出し元に漏えいされるのを防ぎます。また、これらのページ分割トークンは異なる関数間では使用できません。

scannedCount

フィルタ式 (ある場合) が適用される前に、DynamoDB により取得された項目の数です。

例 1

次の例は、GraphQLクエリ: `allPosts` の関数リクエストハンドラーです。

この例では、テーブル内のすべてのエントリが返されます。

```
export function request(ctx) {
  return { operation: 'Scan' };
}
```

例 2

次の例は、GraphQLクエリ: `postsMatching(title: String!)` の関数リクエストハンドラーです。

この例では、タイトルが `title` 引数で始まるテーブル内のすべてのエントリが返されます。

```
export function request(ctx) {
  const { title } = ctx.args;
  const filter = { filter: { beginsWith: title } };
  return {
    operation: 'Scan',
    filter: JSON.parse(util.transform.toDynamoDBFilterExpression(filter)),
  };
}
```

DynamoDB Scan API の詳細については、「DynamoDB API のドキュメント」を参照してください。

同期

Sync リクエストオブジェクトを使用すると、DynamoDB テーブルからすべての結果を取得し、最後のクエリ (差分更新) 以降に変更されたデータのみを受け取ることができます。Sync リクエストは、バージョン管理された DynamoDB データソースに対してのみ実行できます。以下を指定することができます。

- 結果を除外するフィルタ
- 返す項目の数
- ページ分割トークン
- 最後の Sync オペレーションが開始された日時

Sync リクエストオブジェクトのノードの構造は次のとおりです。

```
type DynamoDBSyncRequest = {
  operation: 'Sync';
  basePartitionKey?: string;
  deltaIndexName?: string;
  limit?: number;
  nextToken?: string;
  lastSync?: number;
  filter?: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
};
```

各フィールドの定義は以下のようになります。

Sync フィールド

Sync フィールドリスト

`operation`

実行する DynamoDB の処理。Sync の処理を実行するには、これに Sync を設定する必要があります。この値は必須です。

`filter`

DynamoDB からの結果が返される前に、その結果をフィルタリングするために使用するフィルタです。フィルタの詳細については、「フィルタ」を参照してください。このフィールドはオプションです。

`limit`

一度に評価する項目の最大数です。このフィールドはオプションです。省略した場合、デフォルトの制限は `100` 項目に設定されます。このフィールドの最大値は `1000` 項目です。

`nextToken`

前のクエリを継続するためのページ分割トークンです。これは前のクエリから取得されます。このフィールドはオプションです。

`lastSync`

最後に成功した Sync オペレーションが開始されたエポックミリ秒単位の時刻。指定すると、`lastSync` 以降に変更された項目のみが返されます。このフィールドはオプションです。最初の Sync オペレーションからすべてのページを取得した後にのみ入力する必要があります。省略した場合は、ベーステーブルの結果が返されます。それ以外の場合は、差分テーブルの結果が返されます。

`basePartitionKey`

Sync オペレーションを実行する際に使用されるベーステーブルのパーティションキー。このフィールドにより、テーブルがカスタムパーティションキーを使用している場合に Sync オペレーションを実行できます。これはオプションのフィールドです。

`deltaIndexName`

Sync オペレーションに使用されるインデックス。このインデックスは、テーブルがカスタムパーティションキーを使用する場合に、デルタストアテーブル全体で Sync オペレーション作を

有効にするために必要です。Sync オペレーションは GSI (gsi_ds_pk と gsi_ds_sk で作成) 上で実行されます。このフィールドはオプションです。

DynamoDB 同期により返された結果が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、コンテキスト結果 (context.result) で参照できます。

DynamoDB の型変換の詳細については、「型システム (リクエストマッピング)」を参照してください。

JavaScript リゾルバーの詳細については、「JavaScript リゾルバーの概要」を参照してください。

結果は以下の構造を持ちます。

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10,
    startedAt = 1550000000000
}
```

各フィールドの定義は以下のようになります。

items

同期により返された項目を含むリストです。

nextToken

さらに結果がある場合、 には別のリクエストで使用できるページ分割トークンnextTokenが含まれています。 AWS AppSync は、DynamoDB から返されたページ分割トークンを暗号化して難読化します。これにより、テーブルデータが誤って呼び出し元に漏えいされるのを防ぎます。また、これらのページ分割トークンは異なる関数間では使用できません。

scannedCount

フィルタ式 (ある場合) が適用される前に、DynamoDB により取得された項目の数です。

startedAt

エポックミリ秒単位の時刻ですが、同期オペレーションが開始されれば、ローカルに保存して別のリクエストで lastSync の引数として使用することができます。ページ分割トークンがリクエストに含まれている場合、この値は、結果の最初のページのリクエストによって返されたものと同じになります。

例

次の例は、GraphQLクエリ: `syncPosts(nextToken: String, lastSync: AWSTimestamp)` の関数リクエストハンドラーです。

この例では、`lastSync` を省略すると、ベーステーブルのすべてのエントリが返されます。`lastSync` が指定されている場合は、`lastSync` 以降に変更された差分同期テーブルのエントリのみが返されます。

```
export function request(ctx) {
  const { nextToken, lastSync } = ctx.args;
  return { operation: 'Sync', limit: 100, nextToken, lastSync };
}
```

BatchGetItem

`BatchGetItem` リクエストオブジェクトを使用すると、 AWS AppSync DynamoDB 関数に、複数のテーブルにまたがる可能性のある複数の項目を取得するリクエストを DynamoDB `BatchGetItem`に行うように指示できます。このリクエストオブジェクトでは、以下の情報を指定する必要があります。

- 項目を取得するテーブルの名前
- 各テーブルから取得する項目のキー

DynamoDB の `BatchGetItem` 制限が適用されるため、条件式を指定することはできません。

`BatchGetItem` リクエストオブジェクトのノードの構造は次のとおりです。

```
type DynamoDBBatchGetItemRequest = {
  operation: 'BatchGetItem';
  tables: {
    [tableName: string]: {
      keys: { [key: string]: any }[];
      consistentRead?: boolean;
      projection?: {
        expression: string;
        expressionNames?: { [key: string]: string };
      };
    };
  };
```

```
};
```

各フィールドの定義は以下のようになります。

BatchGetItem フィールド

BatchGetItem フィールドリスト

`operation`

実行する DynamoDB の処理。`BatchGetItem` DynamoDB の処理を実行するには、これを `BatchGetItem` に設定する必要があります。この値は必須です。

`tables`

項目を取得する DynamoDB テーブル。値は、キーとしてテーブル名が指定されているマップです。1 つ以上のテーブルを指定する必要があります。この `tables` の値は必須です。

`keys`

取り出す項目のプライマリキーを表す DynamoDB キーのリスト。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。

`consistentRead`

GetItem 処理の実行時に整合性のある読み込みを使用するかどうか。この値はオプションであり、デフォルトは false です。

`projection`

DynamoDB オペレーションから返される属性を指定するために使用されるプロジェクション。プロジェクションの詳細については、「プロジェクション」を参照してください。このフィールドはオプションです。

覚えておくべきポイント:

- 項目がテーブルから取得されなかった場合は、そのテーブルの data ブロックに null 要素があります。
- 呼び出し結果は、リクエストオブジェクト内で提供された順序に基づいて、テーブル別にソートされます。

- BatchGetItem 内の各 Get コマンドはアトミックですが、バッチは部分的に処理される場合があります。エラーのためにバッチが部分的に処理された場合、未処理のキーは unprocessedKeys ブロック内に呼び出し結果の一部として返されます。
- BatchGetItem は 100 キーに制限されています。

次の例の関数リクエストハンドラーの場合:

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'BatchGetItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId })],
      posts: [util.dynamodb.toMapValues({ authorId, postId })],
    },
  };
}
```

ctx.result で使用可能な呼び出し結果は以下のとおりです。

```
{
   "data": {
     "authors": [null],
     "posts": [
        // Was retrieved
        {
          "authorId": "a1",
          "postId": "p2",
          "postTitle": "title",
          "postDescription": "description",
        }
     ]
   },
   "unprocessedKeys": {
     "authors": [
        // This item was not processed due to an error
        {
          "authorId": "a1"
        }
```

```
      ],
      "posts": []
    }
}
```

ctx.error に エラーに関する詳細が含まれています。data キー、unprocessedKeys キー、および リクエストマッピングテンプレートで渡された各テーブルキーは呼び出し結果に必ずあります。削除された項目は data ブロックにあります。処理されなかった項目は、data ブロック内で null としてマークされ、unprocessedKeys ブロックに挿入されます。

BatchDeleteItem

BatchDeleteItem リクエストオブジェクトを使用すると、 AWS AppSync DynamoDB 関数に、複数のテーブルにまたがる可能性のある複数の項目を削除するリクエストを DynamoDB BatchWriteItemに行うように指示できます。このリクエストオブジェクトでは、以下の情報を指定する必要があります。

- 項目を削除するテーブルの名前
- 各テーブルから削除する項目のキー

DynamoDB の BatchWriteItem 制限が適用されるため、条件式を指定することはできません。

BatchDeleteItem リクエストオブジェクトのノードの構造は次のとおりです。

```
type DynamoDBBatchDeleteItemRequest = {
  operation: 'BatchDeleteItem';
  tables: {
    [tableName: string]: { [key: string]: any }[];
  };
};
```

各フィールドの定義は以下のようになります。

BatchDeleteItem フィールド

BatchDeleteItem フィールドリスト

operation

実行する DynamoDB の処理。BatchDeleteItem DynamoDB の処理を実行するには、これを BatchDeleteItem に設定する必要があります。この値は必須です。

`tables`

項目を削除する DynamoDB テーブル。各テーブルは、削除する項目のプライマリキーを表す DynamoDB キーのリストです。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。1 つ以上のテーブルを指定する必要があります。値 が必要です。

覚えておくべきポイント:

- `DeleteItem` オペレーションとは対照的に、完全に削除された項目はレスポンスで返されません。渡されたキーのみが返されます。
- 項目がテーブルから削除されなかった場合は、そのテーブルの data ブロックに null 要素があります。
- 呼び出し結果は、リクエストオブジェクト内で提供された順序に基づいて、テーブル別にソートされます。
- `Delete` 内部の各 `BatchDeleteItem` コマンドはアトミックです。ただし、バッチは部分的に処理できます。エラーのためにバッチが部分的に処理された場合、未処理のキーは unprocessedKeys ブロック内に呼び出し結果の一部として返されます。
- `BatchDeleteItem` は 25 キーに制限されています。
- このオペレーション、競合検出で使用するとサポートされていません。両方を同時に使用すると、エラーが発生する可能性があります。

次の例の関数リクエストハンドラーの場合:

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'BatchDeleteItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId })],
      posts: [util.dynamodb.toMapValues({ authorId, postId })],
    },
  };
}
```

ctx.result で使用可能な呼び出し結果は以下のとおりです。

```
{
   "data": {
     "authors": [null],
     "posts": [
        // Was deleted
        {
          "authorId": "a1",
          "postId": "p2"
        }
     ]
   },
   "unprocessedKeys": {
     "authors": [
        // This key was not processed due to an error
        {
          "authorId": "a1"
        }
      ],
     "posts": []
   }
}
```

ctx.error にエラーに関する詳細が含まれています。data キー、unprocessedKeys キー、および関数リクエストオブジェクトで渡された各テーブルキーは呼び出し結果に必ずあります。削除された項目は data ブロックにあります。処理されなかった項目は、data ブロック内で null としてマークされ、unprocessedKeys ブロックに挿入されます。

BatchPutItem

BatchPutItem リクエストオブジェクトを使用すると、 AWS AppSync DynamoDB 関数に、複数のテーブルにまたがる可能性のある複数の項目を配置するように DynamoDB にBatchWriteItemリクエストするように指示できます。このリクエストオブジェクトでは、以下の情報を指定する必要があります。

- 項目を挿入するテーブルの名前
- 各テーブルに挿入するすべての項目

DynamoDB の BatchWriteItem 制限が適用されるため、条件式を指定することはできません。

BatchPutItem リクエストオブジェクトのノードの構造は次のとおりです。

```
type DynamoDBBatchPutItemRequest = {
  operation: 'BatchPutItem';
  tables: {
    [tableName: string]: { [key: string]: any}[];
  };
};
```

各フィールドの定義は以下のようになります。

BatchPutItem フィールド

BatchPutItem フィールドリスト

operation

実行する DynamoDB の処理。BatchPutItem DynamoDB の処理を実行するには、これを BatchPutItem に設定する必要があります。この値は必須です。

tables

項目を挿入する DynamoDB テーブル。各テーブルエントリは、この特定のテーブルに挿入する DynamoDB 項目のリストを表します。1 つ以上のテーブルを指定する必要があります。この値は必須です。

覚えておくべきポイント:

- 完全に挿入された項目がレスポンスに返されます (正常に挿入された場合)。
- 項目がテーブルに挿入されなかった場合は、そのテーブルの data ブロックに null 要素があります。
- 挿入された項目は、リクエストオブジェクト内で提供された順序に基づいて、テーブル別にソートされます。
- Put 内の各 BatchPutItem コマンドはアトミックですが、バッチは部分的に処理される場合があります。エラーのためにバッチが部分的に処理された場合、未処理のキーは unprocessedKeys ブロック内に呼び出し結果の一部として返されます。
- BatchPutItem は 25 項目に制限されています。
- このオペレーション、競合検出で使用するとサポートされていません。両方を同時に使用すると、エラーが発生する可能性があります。

次の例の関数リクエストハンドラーの場合:

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId, name, title } = ctx.args;
  return {
    operation: 'BatchPutItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId, name })],
      posts: [util.dynamodb.toMapValues({ authorId, postId, title })],
    },
  };
}
```

`ctx.result` で使用可能な呼び出し結果は以下のとおりです。

```
{
   "data": {
     "authors": [
         null
     ],
     "posts": [
        // Was inserted
        {
          "authorId": "a1",
          "postId": "p2",
          "title": "title"
        }
     ]
   },
   "unprocessedItems": {
     "authors": [
        // This item was not processed due to an error
        {
          "authorId": "a1",
          "name": "a1_name"
        }
      ],
     "posts": []
   }
}
```

ctx.error にエラーに関する詳細が含まれています。data キー、The keys data, unprocessedItems キー、およびリクエストオブジェクトで渡された各テーブルキーは呼び出し結果に必ずあります。挿入された項目は data ブロックにあります。処理されなかった項目は、data ブロック内で null としてマークされ、unprocessedItems ブロックに挿入されます。

TransactGetItems

TransactGetItems リクエストオブジェクトを使用すると、 AWS AppSync DynamoDB 関数に、複数のテーブルにまたがる可能性のある複数の項目を取得するリクエストを DynamoDB TransactGetItemsに行うように指示できます。このリクエストオブジェクトでは、以下の情報を指定する必要があります。

- 項目の取得元となる各リクエスト項目のテーブル名
- 各テーブルから取得する各リクエスト項目のキー

DynamoDB の TransactGetItems 制限が適用されるため、条件式を指定することはできません。

TransactGetItems リクエストオブジェクトのノードの構造は次のとおりです。

```
type DynamoDBTransactGetItemsRequest = {
  operation: 'TransactGetItems';
  transactItems: { table: string; key: { [key: string]: any }; projection?:
 { expression: string; expressionNames?: { [key: string]: string }; }[];
  };
};
```

各フィールドの定義は以下のようになります。

TransactGetItems フィールド

TransactGetItems フィールドリスト

operation

実行する DynamoDB の処理。TransactGetItems DynamoDB の処理を実行するには、これを TransactGetItems に設定する必要があります。この値は必須です。

transactItems

含めるリクエスト項目。値はリクエスト項目の配列です。少なくとも 1 つのリクエスト項目を指定する必要があります。この transactItems の値は必須です。

`table`

項目の取得元となる DynamoDB テーブル。値はテーブル名の文字列です。この `table` の値は必須です。

`key`

取り出す項目のプライマリキーを表す DynamoDB キー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。

`projection`

DynamoDB オペレーションから返される属性を指定するために使用されるプロジェクション。プロジェクションの詳細については、「プロジェクション」を参照してください。このフィールドはオプションです。

覚えておくべきポイント:

- トランザクションが成功すると、`items` ブロック内で取得された項目の順序はリクエスト項目の順序と同じになります。
- トランザクションは、オールオアナッシング方式で実行されます。いずれかのリクエスト項目でエラーが発生した場合、トランザクション全体は実行されず、エラーの詳細が返されます。
- リクエスト項目を取得できなくても、エラーではありません。代わりに、null要素が対応する位置の項目ブロックに表示されます。
- トランザクションのエラーが TransactionCanceledException である場合、`cancellationReasons` ブロックに入力されます。`cancellationReasons` ブロック内のキャンセル理由の順序は、リクエスト項目の順序と同じになります。
- `TransactGetItems` のリクエスト項目数は 100 個に制限されています。

次の例の関数リクエストハンドラーの場合:

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'TransactGetItems',
    transactItems: [
```

```
      {
        table: 'posts',
        key: util.dynamodb.toMapValues({ postId }),
      },
      {
        table: 'authors',
        key: util.dynamodb.toMapValues({ authorId }),
      },
    ],
  };
}
```

トランザクションが成功し、最初にリクエストされた項目だけが取得された場合、ctx.result で使用できる呼び出し結果は次のようになります。

```
{
    "items": [
       {
           // Attributes of the first requested item
           "post_id": "p1",
           "post_title": "title",
           "post_description": "description"
       },
       // Could not retrieve the second requested item
       null,
    ],
    "cancellationReasons": null
}
```

最初のリクエスト項目によって発生した TransactionCanceledException が原因でトランザクションが失敗した場合、ctx.result で使用可能な呼び出し結果は次のようになります。

```
{
    "items": null,
    "cancellationReasons": [
       {
           "type":"Sample error type",
           "message":"Sample error message"
       },
       {
           "type":"None",
           "message":"None"
```

```
        }
    ]
}
```

ctx.error にエラーに関する詳細が含まれています。キー items と cancellationReasons は、ctx.result にあることが保証されています。

TransactWriteItems

TransactWriteItems リクエストオブジェクトを使用すると、 AWS AppSync DynamoDB 関数に、複数のテーブルTransactWriteItemsに複数の項目を書き込むリクエストを DynamoDB に行うように指示できます。このリクエストオブジェクトでは、以下の情報を指定する必要があります。

- 各リクエスト項目の書き込み先テーブル名
- 実行する各リクエスト項目のオペレーション。サポートされているオペレーションには、PutItem、UpdateItem、DeleteItem、および ConditionCheck の 4 種類があります。
- 書き込む各リクエスト項目のキー

DynamoDB の TransactWriteItems 制限が適用されます。

TransactWriteItems リクエストオブジェクトのノードの構造は次のとおりです。

```
type DynamoDBTransactWriteItemsRequest = {
  operation: 'TransactWriteItems';
  transactItems: TransactItem[];
};
type TransactItem =
  | TransactWritePutItem
  | TransactWriteUpdateItem
  | TransactWriteDeleteItem
  | TransactWriteConditionCheckItem;
type TransactWritePutItem = {
  table: string;
  operation: 'PutItem';
  key: { [key: string]: any };
  attributeValues: { [key: string]: string};
  condition?: TransactConditionCheckExpression;
};
type TransactWriteUpdateItem = {
  table: string;
  operation: 'UpdateItem';
```

```
  key: { [key: string]: any };
  update: DynamoDBExpression;
  condition?: TransactConditionCheckExpression;
};
type TransactWriteDeleteItem = {
  table: string;
  operation: 'DeleteItem';
  key: { [key: string]: any };
  condition?: TransactConditionCheckExpression;
};
type TransactWriteConditionCheckItem = {
  table: string;
  operation: 'ConditionCheck';
  key: { [key: string]: any };
  condition?: TransactConditionCheckExpression;
};
type TransactConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
  returnValuesOnConditionCheckFailure: boolean;
};
```

TransactWriteItems フィールド

TransactWriteItems フィールドリスト

各フィールドの定義は以下のようになります。

`operation`

実行する DynamoDB の処理。`TransactWriteItems` DynamoDB の処理を実行するには、これを `TransactWriteItems` に設定する必要があります。この値は必須です。

`transactItems`

含めるリクエスト項目。値はリクエスト項目の配列です。少なくとも 1 つのリクエスト項目を指定する必要があります。この `transactItems` の値は必須です。

`PutItem` の場合、各フィールドの定義は以下のようになります。

`table`

送信先のの DynamoDB テーブル。値はテーブル名の文字列です。この `table` の値は必須です。

operation

実行する DynamoDB の処理。PutItem DynamoDB の処理を実行するには、これを PutItem に設定する必要があります。この値は必須です。

key

配置する項目のプライマリキーを表す DynamoDB キー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

attributeValues

DynamoDB に渡す項目の残りの属性です。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。このフィールドはオプションです。

condition

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件が指定されていない場合、PutItem リクエストはその項目の既存のエントリを上書きします。条件チェックに失敗した場合に、既存の項目を取得し直すかどうかを指定できます。トランザクション条件の詳細については、「トランザクション条件式」を参照してください。この値はオプションです。

UpdateItem の場合、各フィールドの定義は以下のようになります。

table

更新する DynamoDB テーブル。値はテーブル名の文字列です。この table の値は必須です。

operation

実行する DynamoDB の処理。UpdateItem DynamoDB の処理を実行するには、これを UpdateItem に設定する必要があります。この値は必須です。

key

更新する項目のプライマリキーを表す DynamoDB キー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

update

update セクションには、 DynamoDBの項目の更新方法を示す更新式を指定することができます。更新式の記述方法の詳細については、「DynamoDB UpdateExpressions」のドキュメントを参照してください。このセクションは必須です。

condition

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件を指定していない場合は、UpdateItem リクエストによって、現在の状態にかかわらず、既存のエントリが更新されます。条件チェックに失敗した場合に、既存の項目を取得し直すかどうかを指定できます。トランザクション条件の詳細については、「トランザクション条件式」を参照してください。この値はオプションです。

DeleteItem の場合、各フィールドの定義は以下のようになります。

table

項目を削除する DynamoDB テーブル。値はテーブル名の文字列です。この table の値は必須です。

operation

実行する DynamoDB の処理。DeleteItem DynamoDB の処理を実行するには、これを DeleteItem に設定する必要があります。この値は必須です。

key

削除する項目のプライマリキーを表す DynamoDB キー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

condition

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件を指定していない場合、DeleteItem リクエストによって、現在の状態にかかわらず、項目が削除されます。条件チェックに失敗した場合に、既存の項目を取得し直すかどうかを指定できます。トランザクション条件の詳細については、「トランザクション条件式」を参照してください。この値はオプションです。

ConditionCheck の場合、各フィールドの定義は以下のようになります。

`table`

条件をチェックする DynamoDB テーブル。値はテーブル名の文字列です。この `table` の値は必須です。

`operation`

実行する DynamoDB の処理。`ConditionCheck` DynamoDB の処理を実行するには、これを `ConditionCheck` に設定する必要があります。この値は必須です。

`key`

条件チェックする項目のプライマリキーを表す DynamoDB キー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

`condition`

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件チェックに失敗した場合に、既存の項目を取得し直すかどうかを指定できます。トランザクション条件の詳細については、「トランザクション条件式」を参照してください。この値は必須です。

覚えておくべきポイント:

- リクエスト項目のキーのみがレスポンスに返されます (正常に挿入された場合)。キーの順序は、リクエスト項目の順序と同じです。
- トランザクションは、オールオアナッシング方式で実行されます。いずれかのリクエスト項目でエラーが発生した場合、トランザクション全体は実行されず、エラーの詳細が返されます。
- 同じ項目をターゲットにできるリクエスト項目が 2 つありません。それ以外の場合は、TransactionCanceledException エラーが発生します。
- トランザクションのエラーが TransactionCanceledException である場合、`cancellationReasons` ブロックに入力されます。リクエスト項目の条件チェックが失敗し、かつ `returnValuesOnConditionCheckFailure` を `false` に指定しなかった場合、テーブルに存在する項目が取得され、`item` で `cancellationReasons` ブロックの対応する位置に保存されます。
- `TransactWriteItems` のリクエスト項目数は 100 個に制限されています。
- このオペレーション、競合検出で使用するとサポートされていません。両方を同時に使用すると、エラーが発生する可能性があります。

次の例の関数リクエストハンドラーの場合:

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId, title, description, oldTitle, authorName } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({ postId }),
        attributeValues: util.dynamodb.toMapValues({ title, description }),
        condition: util.transform.toDynamoDBConditionExpression({
          title: { eq: oldTitle },
        }),
      },
      {
        table: 'authors',
        operation: 'UpdateItem',
        key: util.dynamodb.toMapValues({ authorId }),
        update: {
          expression: 'SET authorName = :name',
          expressionValues: util.dynamodb.toMapValues({ ':name': authorName }),
        },
      },
    ],
  };
}
```

トランザクションが成功した場合、ctx.result で使用できる呼び出し結果は次のようになります。

```
{
    "keys": [
       // Key of the PutItem request
       {
           "post_id": "p1",
       },
       // Key of the UpdateItem request
       {
           "author_id": "a1"
```

```
        }
    ],
    "cancellationReasons": null
}
```

PutItem リクエストの条件チェックに失敗したためにトランザクションが失敗した場合、 で使用できる呼び出し結果は次のようになります。

```
{
    "keys": null,
    "cancellationReasons": [
        {
            "item": {
                "post_id": "p1",
                "post_title": "Actual old title",
                "post_description": "Old description"
            },
            "type": "ConditionCheckFailed",
            "message": "The condition check failed."
        },
        {
            "type": "None",
            "message": "None"
        }
    ]
}
```

ctx.error にエラーに関する詳細が含まれています。keys および cancellationReasons が ctx.result に存在することが保証されています。

型システム (リクエストマッピング)

AWS AppSync DynamoDB 関数を使用して DynamoDB テーブルを呼び出す場合、 AWS AppSync はその呼び出しで使用する各値のタイプを知る必要があります。これは、DynamoDB が GraphQL や JSON よりも多くのタイプのプリミティブ (セットやバイナリデータなど) をサポートしているためです。 AWS AppSync は、GraphQL と DynamoDB の間で変換するときにいくつかのヒントを必要とします。そうしないと、テーブル内のデータの構造についていくつかの仮定を行う必要があります。

DynamoDB のデータ型の詳細については、DynamoDB の「データ型記述子」および「データ型」の各ドキュメントを参照してください。

DynamoDB の値は、単一のキーと値のペアを含む JSON オブジェクトで表されます。キーは DynamoDB の型を指定し、値はその値自身を指定します。次の例では、キー S は値が文字列であることを示し、値 identifier がその文字列値です。

```
{ "S" : "identifier" }
```

JSON オブジェクトは複数のキーと値のペアを持つことはできません。複数のキーと値のペアが指定されている場合、リクエストオブジェクトは解析されません。

DynamoDB の値は、値を指定する必要がある場合にはリクエストオブジェクトのどこかで使用されます。これが必要になる箇所には、key セクションと attributeValue セクション、および式セクションの expressionValues セクションが含まれています。次の例では、DynamoDB の文字列値 identifier が、(おそらくは GetItem リクエストオブジェクトの) key セクションの id フィールドに割り当てられています。

```
"key" : {
   "id" : { "S" : "identifier" }
}
```

サポートされているタイプ

AWS AppSync は、次の DynamoDB スカラー、ドキュメント、およびセットタイプをサポートしています。

文字列型 **S**

単一の文字列値です。DynamoDB の文字列値は次のように表されます。

```
{ "S" : "some string" }
```

以下は使用例です。

```
"key" : {
   "id" : { "S" : "some string" }
}
```

文字列セット型 **SS**

1 組の文字列値です。DynamoDB の文字列セット値は次のように表されます。

```
{ "SS" : [ "first value", "second value", ... ] }
```

以下は使用例です。

```
"attributeValues" : {
   "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
}
```

数値型 **N**

単一の数値です。DynamoDB の数値は次のように表されます。

```
{ "N" : 1234 }
```

以下は使用例です。

```
"expressionValues" : {
   ":expectedVersion" : { "N" : 1 }
}
```

数値セット型 **NS**

1 組の数値です。DynamoDB の数値セット値は次のように表されます。

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

以下は使用例です。

```
"attributeValues" : {
   "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }
}
```

バイナリ型 **B**

バイナリ値です。DynamoDB のバイナリ値は次のように表されます。

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

値は実際には文字列であり、文字列はバイナリデータの base64 でエンコードされた表現です。AWS AppSync はこの文字列をバイナリ値にデコードしてから DynamoDB に送信します。 AWS AppSync は RFC 2045 で定義されている base64 デコードスキームを使用します。base64 アルファベットにない文字は無視されます。

以下は使用例です。

```
"attributeValues" : {
   "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

バイナリセット型 **BS**

1 組のバイナリ値です。DynamoDB のバイナリセット値は次のように表されます。

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

値は実際には文字列であり、文字列はバイナリデータの base64 でエンコードされた表現です。AWS AppSync はこの文字列をバイナリ値にデコードしてから DynamoDB に送信します。 AWS AppSync は RFC 2045 で定義されている base64 デコードスキームを使用します。base64 アルファベットにない文字は無視されます。

以下は使用例です。

```
"attributeValues" : {
   "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }
}
```

ブール型 **BOOL**

ブール値。DynamoDB のブール値は次のように表されます。

```
{ "BOOL" : true }
```

有効な値は、`true` と `false` のみです。

以下は使用例です。

```
"attributeValues" : {
```

```
   "orderComplete" : { "BOOL" : false }
}
```

リスト型 L

サポートされているその他の DynamoDB の値のリストです。DynamoDB のリスト値は次のように表されます。

```
{ "L" : [ ... ] }
```

この値は複合値です。リストには、サポートされる DynamoDB の値 (他のリストも含む) が 0 個以上入ります。このリストには、異なる型を混在させることもできます。

以下は使用例です。

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

マップ型 M

サポートされる他の DynamoDB の値の、キーと値のペアの順序付けされていない集合を表します。DynamoDB のマップ値は次のように表されます。

```
{ "M" : { ... } }
```

マップには 0 個以上のキーと値のペアが入ります。キーは文字列である必要があり、値には、サポートされている DynamoDB の任意の値 (他のマップを含む) が使用できます。このマップには、異なる型を混在させることもできます。

以下は使用例です。

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
```

```
    }
}
```

Null 型 **NULL**

null 値です。DynamoDB の Null 値は次のように表されます。

```
{ "NULL" : null }
```

以下は使用例です。

```
"attributeValues" : {
   "phoneNumbers" : { "NULL" : null }
}
```

各タイプの詳細については、DynamoDB のドキュメントを参照してください。

型システム (レスポンスマッピング)

DynamoDB からレスポンスを受信すると、 AWS AppSync はそれを GraphQL プリミティブ型と JSON プリミティブ型に自動的に変換します。DynamoDB の各属性はデコードされ、レスポンスハンドラーのコンテキストで返されます。

たとえば、DynamoDB が以下を返したとします。

```
{
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

パイプラインリゾルバーから結果が返されると、 AWS AppSync はそれを次のように GraphQL および JSON タイプに変換します。

```
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

このセクションでは、 AWS AppSync が次の DynamoDB スカラー、ドキュメント、およびセットタイプを変換する方法について説明します。

文字列型 **S**

単一の文字列値です。DynamoDB 文字列値が文字列として返されます。

たとえば、DynamoDB が次の DynamoDB の文字列値を返したとします。

```
{ "S" : "some string" }
```

AWS AppSync はこれを文字列に変換します。

```
"some string"
```

文字列セット型 **SS**

1 組の文字列値です。DynamoDB 文字列セット値が文字列のリストとして返されます。

たとえば、DynamoDB が次の DynamoDB 文字列セット値を返したとします。

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync はこれを文字列のリストに変換します。

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

数値型 **N**

単一の数値です。DynamoDB 数値が数字として返されます。

たとえば、DynamoDB が次の DynamoDB の数値を返したとします。

```
{ "N" : 1234 }
```

AWS AppSync はこれを数値に変換します。

```
1234
```

数値セット型 **NS**

1 組の数値です。DynamoDB 数値セットが数字のリストとして返されます。

たとえば、DynamoDB が次の DynamoDB の数値セット値を返したとします。

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync はこれを数値のリストに変換します。

```
[ 67.8, 12.2, 70 ]
```

バイナリ型 **B**

バイナリ値です。DynamoDB のバイナリ値は、その値を Base64 で表した文字列として返されます。

たとえば、DynamoDB が次の DynamoDB のバイナリ値を返したとします。

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync は、値を base64 で表現した文字列に変換します。

```
"SGVsbG8sIFdvcmxkIQo="
```

バイナリデータは、RFC 4648 と RFC 2045 で指定されているようにして Base64 エンコーディングスキームにエンコードされます。

バイナリセット型 **BS**

1 組のバイナリ値です。DynamoDB のバイナリセット値は、その値を Base64 で表した文字列のリストとして返されます。

たとえば、DynamoDB が次の DynamoDB のバイナリ値を返したとします。

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync は、値を base64 で表現した文字列のリストに変換します。

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

バイナリデータは、RFC 4648 と RFC 2045 で指定されているようにして Base64 エンコーディングスキームにエンコードされます。

ブール型 **BOOL**

ブール値。DynamoDB ブール値がブールとして返されます。

たとえば、DynamoDB が次の DynamoDB のブール値を返したとします。

```
{ "BOOL" : true }
```

AWS AppSync はこれをブール値に変換します。

```
true
```

リスト型 **L**

サポートされているその他の DynamoDB の値のリストです。DynamoDB のリスト値は値のリストとして返され、内部のそれぞれの値も変換されます。

たとえば、DynamoDB が次の DynamoDB の文字列値を返したとします。

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

AWS AppSync は、変換された値のリストに変換します。

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

マップ型 **M**

サポートされるその他の DynamoDB の値のキー/値の集合です。DynamoDB のマップ値は JSON オブジェクトとして返されます。それぞれのキー/値も変換されます。

たとえば、DynamoDB が次の DynamoDB のバイナリ値を返したとします。

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

AWS AppSync はこれを JSON オブジェクトに変換します。

```
{
   "someString" : "A string value",
   "someNumber" : 1,
   "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

Null 型 **NULL**

null 値です。

たとえば、DynamoDB が次の DynamoDB のブール値を返したとします。

```
{ "NULL" : null }
```

AWS AppSync はこれを null に変換します。

```
null
```

フィルター

Query 処理と Scan 処理を使用して DynamoDB のオブジェクトにクエリを実行する場合、オプションで、結果を評価する filter を指定して、必要な値のみを返すことができます。

Query または Scan リクエストの filter プロパティの構造は以下のとおりです。

```
type DynamoDBExpression = {
```

```
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
};
```

各フィールドの定義は以下のようになります。

expression

クエリ式です。フィルタ式の記述方法の詳細については、「DynamoDB QueryFilter」および「DynamoDB ScanFilter」の各ドキュメントを参照してください。このフィールドの指定は必須です。

expressionNames

式の属性名のプレースホルダーを示します。キー - 値のペアの形式になります。キーは、expression で使用される名前のプレースホルダーに対応します。値は、DynamoDB の項目の属性名に対応する文字列である必要があります。このフィールドはオプションであり、expression で使用される式の属性名のプレースホルダーのみを入力します。

expressionValues

式の属性値のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される値のプレースホルダーに対応し、値は型付き値でなければなりません。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この指定は必須です。このフィールドはオプションであり、expression で使用される式の属性値のプレースホルダーのみを入力します。

例

次の例は、リクエストのフィルターセクションです。ここでは、DynamoDB から取得されたエントリのうち、タイトルが title 引数で始まるもののみが返されます。

ここでは、util.transform.toDynamoDBFilterExpression を使用してオブジェクトから自動的にフィルターを作成します。

```
const filter = util.transform.toDynamoDBFilterExpression({
  title: { beginsWith: 'far away' },
});

const request = {};
```

```
request.filter = JSON.parse(filter);
```

これにより、以下のフィルターが生成されます。

```
{
  "filter": {
    "expression": "(begins_with(#title,:title_beginsWith))",
    "expressionNames": { "#title": "title" },
    "expressionValues": {
      ":title_beginsWith": { "S": "far away" }
    }
  }
}
```

条件式

PutItem、UpdateItem、および DeleteItem の各 DynamoDB 処理を使用して DynamoDB のオブジェクトをミューテーションする場合、オプションで、処理を実行する前に、DynamoDB にある既存のオブジェクトの状態に基づいてリクエストが成功するかどうかを制御する条件式を指定することができます。

AWS AppSync DynamoDB 関数を使用すると、PutItem、、UpdateItemおよび DeleteItemリクエストオブジェクトで条件式を指定でき、条件が失敗してオブジェクトが更新されなかった場合に従う戦略も許可されます。

例 1

以下の PutItem リクエストオブジェクトには条件式がありません。その結果、同じキーに対応する項目がすでにある場合でも、項目は DynamoDB に挿入され、それにより既存の項目が上書きされます。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, ...values} = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}
```

例 2

次の PutItem オブジェクトには条件式があります。この場合、同じキーの項目が DynamoDB に存在しない場合のみ処理が成功します。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, ...values} = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues(values),
    condition: { expression: "attribute_not_exists(id)" }
  };
}
```

デフォルトでは、条件チェックが失敗した場合、 AWS AppSync DynamoDB 関数はミューテーションにエラーを提供します。

ただし、 AWS AppSync DynamoDB 関数には、開発者が一般的なエッジケースを処理するのに役立つ いくつかの追加機能があります。

- AWS AppSync DynamoDB 関数は、DynamoDB の現在の値が目的の結果と一致すると判断できる場合、オペレーションは成功したかのように扱われます。
- エラーを返す代わりに、カスタム Lambda 関数を呼び出して、 AWS AppSync DynamoDB 関数が失敗を処理する方法を決定するように関数を設定できます。

これらの詳細については、「条件チェックでのエラーを処理する」セクションを参照してください。

DynamoDB の条件式の詳細については、「DynamoDB ConditionExpressions のドキュメント」を参照してください。

条件を指定する

PutItem、UpdateItem、および DeleteItem の各リクエストオブジェクトはすべて、オプションで condition セクションが指定できます。省略した場合、条件チェックは実行されません。指定した場合、処理が成功するには、条件が true となる必要があります。

condition セクションは以下の構造を持ちます。

```
type ConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
  equalsIgnore?: string[];
  consistentRead?: boolean;
  conditionalCheckFailedHandler?: {
    strategy: 'Custom' | 'Reject';
    lambdaArn?: string;
  };
};
```

以下のフィールドに条件を指定します。

`expression`

更新式そのものを指定します。条件式の記述方法の詳細については、DynamoDB ConditionExpressions のドキュメントを参照してください。このフィールドの指定は必須です。

`expressionNames`

式の属性名のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される名前のプレースホルダーに対応し、値は DynamoDB の項目の属性名と一致する文字列でなければなりません。このフィールドはオプションであり、expression で使用される式の属性名のプレースホルダーのみを入力します。

`expressionValues`

式の属性値のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される値のプレースホルダーに対応し、値は型付き値でなければなりません。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この指定は必須です。このフィールドはオプションであり、expression で使用される式の属性値のプレースホルダーのみを入力します。

残りのフィールドは、 AWS AppSync DynamoDB 関数に条件チェックの失敗を処理する方法を指示します。

`equalsIgnore`

`PutItem` オペレーションの使用時に条件チェックが失敗すると、 AWS AppSync DynamoDB 関数は、DynamoDB の現在ある項目と、書き込みを試みた項目を比較します。これらが同じ場合、

処理は成功として扱われます。equalsIgnore フィールドを使用して、 AWS AppSync がその比較を実行するときに無視する属性のリストを指定できます。たとえば、唯一の違いが version 属性である場合、オペレーションは成功として扱われます。このフィールドはオプションです。

consistentRead

条件チェックが失敗すると、 AWS AppSync は強力な整合性のある読み込みを使用して DynamoDB から項目の現在の値を取得します。このフィールドを使用して、結果整合性のある読み込みを代わりに使用するよう AWS AppSync DynamoDB の関数に指示することができます。このフィールドはオプションであり、デフォルトは true です。

conditionalCheckFailedHandler

このセクションでは、 AWS AppSync DynamoDB 関数が DynamoDB の現在の値と予想される結果を比較した後に、条件チェック DynamoDB の失敗を処理する方法を指定できます。このセクションはオプションです。省略した場合、デフォルトの処理は Reject です。

strategy

AWS AppSync DynamoDB 関数が DynamoDB の現在の値を期待される結果と比較した後に実行する戦略。このフィールドは必須であり、以下を値を設定できます。

Reject

ミューテーションは失敗し、GraphQL レスポンスにエラーが追加されます。

Custom

AWS AppSync DynamoDB 関数は、カスタム Lambda 関数を呼び出して、条件チェックの失敗を処理する方法を決定します。strategy が Custom に設定されている場合、lambdaArn フィールドには、呼び出す Lambda 関数の ARN が含まれている必要があります。

lambdaArn

AWS AppSync DynamoDB の関数が、条件チェックで検出されたエラーを処理する方法を決定するために呼び出す Lambda 関数の ARN です。このフィールドは、strategy が Custom に設定されている場合のみ指定する必要があります。この機能の使用方法の詳細については、「条件チェックでのエラーを処理する」を参照してください。

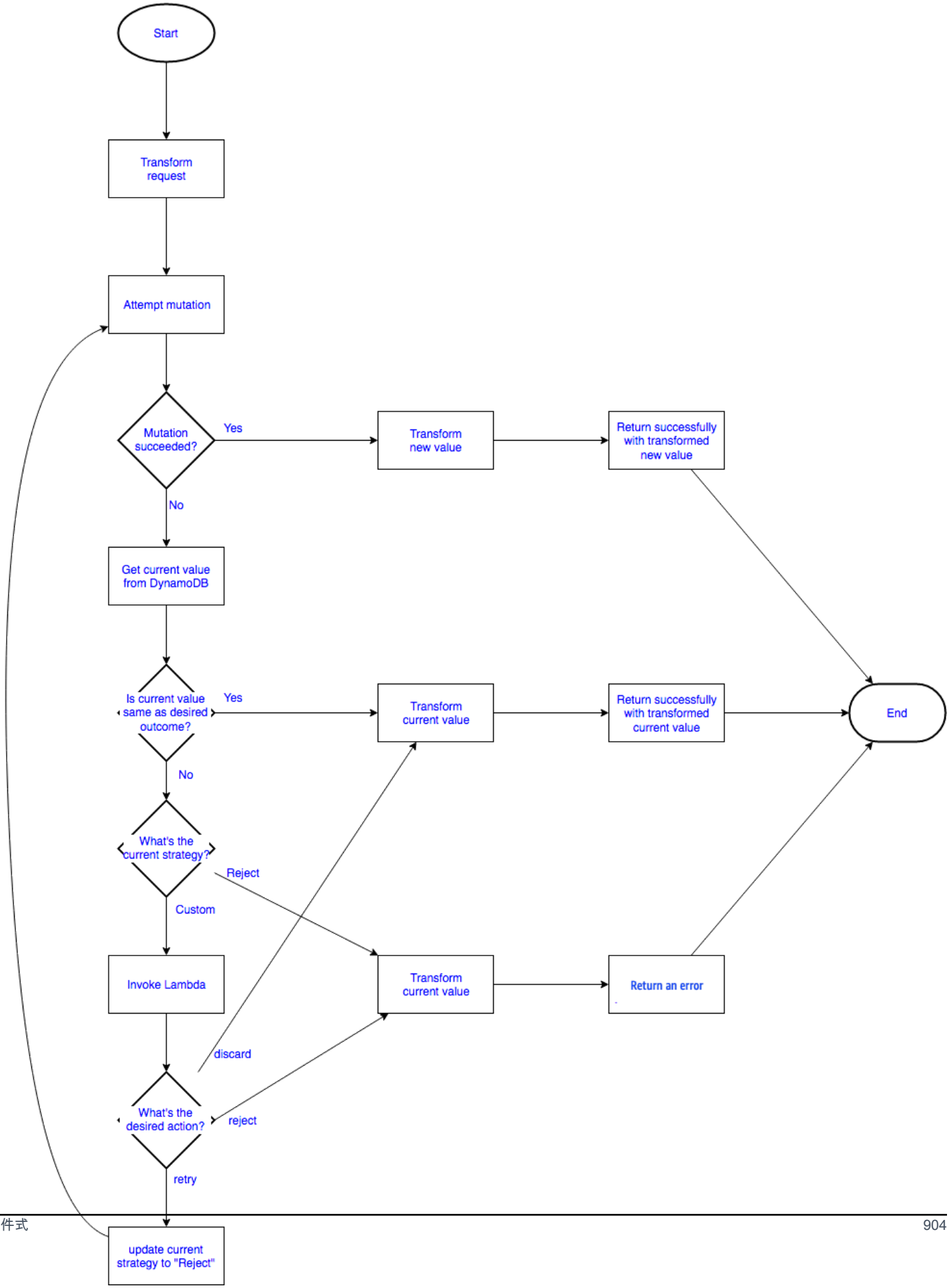
条件チェックでのエラーを処理する

条件チェックが失敗すると、 AWS AppSync DynamoDB 関数は、 util.appendErrorユーティリティを使用して、ミューテーションのエラーとオブジェクトの現在の値を渡すことができます。ただ

し、 AWS AppSync DynamoDB 関数には、開発者が一般的なエッジケースを処理するのに役立つ
くつかの追加機能があります。

- AWS AppSync DynamoDB 関数は、DynamoDB の現在の値が目的の結果と一致すると判断できる場合、オペレーションは成功したかのように扱われます。
- エラーを返す代わりに、カスタム Lambda 関数を呼び出して、 AWS AppSync DynamoDB 関数が失敗を処理する方法を決定するように関数を設定できます。

このプロセスのフローチャートは次のとおりです。


Start
Transform request
Attempt mutation
Mutation succeeded?
Yes
Transform new value
Return successfully with transformed new value
No
Get current value from DynamoDB
Is current value same as desired outcome?
Yes
Transform current value
Return successfully with transformed current value
End
No
What's the current strategy?
Reject
Custom
Invoke Lambda
Transform current value
Return an error
discard
What's the desired action?
reject
retry
update current strategy to "Reject"

必要な結果をチェックする

条件チェックが失敗すると、 AWS AppSync DynamoDB 関数は `GetItem` DynamoDB リクエストを実行して DynamoDB から項目の現在の値を取得します。デフォルトでは、強力な整合性のある読み込みを使用しますが、`condition` ブロックの `consistentRead` フィールドを使用してこれを設定し、この値を期待した結果と比較することができます。

- `PutItem` オペレーションの場合、 AWS AppSync DynamoDB 関数は、現在の値を書き込みを試みた値と比較`equalsIgnore`対象から除外します。項目が同じ場合は、処理は成功として扱われ、DynamoDB から取得された項目が返されます。それ以外の場合は、設定された処理に従います。

  たとえば、`PutItem` リクエストオブジェクトが以下のようになっているとします。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { id, name, version} = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues({ name, version: version+1 }),
    condition: {
      expression: "version = :expectedVersion",
      expressionValues: util.dynamodb.toMapValues({':expectedVersion': version}),
      equalsIgnore: ['version']
    }
  };
}
```

  現在、DynamoDB にある項目は以下のようになりました。

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

  AWS AppSync DynamoDB 関数は、書き込もうとした項目を現在の値と比較し、唯一の違いは `version`フィールドであることを確認できますが、 `version`フィールドを無視するように設定されているため、オペレーションは成功として扱われ、DynamoDB から取得した項目を返します。

- `DeleteItem` オペレーションの場合、 AWS AppSync DynamoDB 関数は項目が DynamoDB から返されたことを確認します。項目が返されなかった場合、処理は成功として扱われます。それ以外の場合は、設定された処理に従います。
- `UpdateItem` オペレーションの場合、 AWS AppSync DynamoDB 関数には、現在 DynamoDB にある項目が期待される結果と一致するかどうかを判断するのに十分な情報がないため、設定された戦略に従います。

DynamoDB のオブジェクトの現在の状態が予想される結果と異なる場合、 AWS AppSync DynamoDB 関数は設定された戦略に従ってミューテーションを拒否するか、Lambda 関数を呼び出して次に何をすべきかを決定します。

「reject」の戦略に従う

`Reject` 戦略に従うと、 AWS AppSync DynamoDB 関数はミューテーションのエラーを返します。

たとえば、次のミューテーションリクエストが指定されたとします。

```
mutation {
    updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
        Name
        theVersion
    }
}
```

DynamoDB から返された項目が以下のようになっているとします。

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

この関数レスポンスハンドラーは次のようになります。

```
import { util } from '@aws-appsync/utils';
export function response(ctx) {
  const { version, ...values } = ctx.result;
  const result = { ...values, theVersion: version };
  if (ctx.error) {
```

```
    if (error) {
      return util.appendError(error.message, error.type, result, null);
    }
  }
  return result
}
```

GraphQL レスポンスは以下のようになります。

```
{
  "data": null,
  "errors": [
    {
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      ...
    }
  ]
}
```

また、返されたオブジェクトのフィールドすべてが他のリゾルバーによって入力され、そのミューテーションが成功した場合、オブジェクトが `error` セクションに返されたときに、それらのフィールドは解決されません。

「custom」戦略に従う

`Custom` 戦略に従うと、 AWS AppSync DynamoDB 関数は Lambda 関数を呼び出して、次に何をすべきかを決定します。Lambda 関数は以下のオプションのいずれかを選択します。

- ミューテーション を `reject` する これにより、 AWS AppSync DynamoDB 関数は、設定された戦略が であるかのように動作し`Reject`、前のセクションで説明したように、ミューテーションのエラーと DynamoDB のオブジェクトの現在の値を返します。
- ミューテーション を `discard` する これにより、 AWS AppSync DynamoDB 関数は条件チェックの失敗を黙って無視し、DynamoDB の値を返します。
- ミューテーション を `retry` する これにより、 AWS AppSync DynamoDB 関数に、新しいリクエストオブジェクトでミューテーションを再試行するように指示します。

Lambda 呼び出しリクエスト

AWS AppSync DynamoDB 関数は、 で指定された Lambda 関数を呼び出しますlambdaArn。また、データソースに設定されたものと同じ service-role-arn を使用します。呼び出しのペイロードは以下の構造を持ちます。

```
{
    "arguments": { ... },
    "requestMapping": {... },
    "currentValue": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

各フィールドの定義は以下のようになります。

arguments

GraphQL ミューテーションの引数です。これは、context.arguments のリクエストオブジェクトで使用できる引数と同じです。

requestMapping

このオペレーションのリクエストオブジェクト。

currentValue

DynamoDB のオブジェクトの現在値。

resolver

AWS AppSync リゾルバーまたは関数に関する情報。

identity

呼び出し元に関する情報。これは、context.identity のリクエストオブジェクトで使用できる識別情報と同じです。

完全なペイロードの例を次に示します。

```
{
    "arguments": {
        "id": "1",
        "name": "Steve",
        "expectedVersion": 1
    },
```

```
    "requestMapping": {
        "version" : "2017-02-28",
        "operation" : "PutItem",
        "key" : {
           "id" : { "S" : "1" }
        },
        "attributeValues" : {
           "name" : { "S" : "Steve" },
           "version" : { "N" : 2 }
        },
        "condition" : {
           "expression" : "version = :expectedVersion",
           "expressionValues" : {
               ":expectedVersion" : { "N" : 1 }
           },
           "equalsIgnore": [ "version" ]
        }
    },
    "currentValue": {
        "id" : { "S" : "1" },
        "name" : { "S" : "Steve" },
        "version" : { "N" : 8 }
    },
    "resolver": {
        "tableName": "People",
        "awsRegion": "us-west-2",
        "parentType": "Mutation",
        "field": "updatePerson",
        "outputType": "Person"
    },
    "identity": {
        "accountId": "123456789012",
        "sourceIp": "x.x.x.x",
        "user": "AIDAAAAAAAAAAAAAAAAA",
        "userArn": "arn:aws:iam::123456789012:user/appsync"
    }
}
```

Lambda 呼び出しレスポンス

Lambda 関数は、呼び出しペイロードを検査し、任意のビジネスロジックを適用して、 AWS AppSync DynamoDB 関数が障害を処理する方法を決定できます。条件チェックで検出された エラーを処理するために、以下の 3 つのオプションが指定できます。

- ミューテーション を `reject` する このオプションのレスポンスペイロードは次の構造を持ちます。

```
{
    "action": "reject"
}
```

これにより、 AWS AppSync DynamoDB 関数は設定された戦略が であるかのように動作しReject、上記のセクションで説明されているように、ミューテーションのエラーと DynamoDB 内のオブジェクトの現在の値を返します。

- ミューテーション を `discard` する このオプションのレスポンスペイロードは次の構造を持ちます。

```
{
    "action": "discard"
}
```

これにより、 AWS AppSync DynamoDB 関数は条件チェックの失敗を黙って無視し、DynamoDB の値を返します。

- ミューテーション を `retry` する このオプションのレスポンスペイロードは次の構造を持ちます。

```
{
    "action": "retry",
    "retryMapping": { ... }
}
```

これにより、 AWS AppSync DynamoDB 関数に、新しいリクエストオブジェクトでミューテーションを再試行するように指示します。`retryMapping` セクションの構造は DynamoDB の処理によって異なり、その処理の完全なリクエストオブジェクトのサブセットとなります。

`PutItem` の場合、`retryMapping` セクションは次の構造を持ちます。`attributeValues` フィールドについては、「PutItem」を参照してください。

```
{
    "attributeValues": { ... },
    "condition": {
        "equalsIgnore" = [ ... ],
```

```
        "consistentRead" = true
    }
}
```

UpdateItem の場合、retryMapping セクションは次の構造を持ちます。update セクションについては、「UpdateItem」を参照してください。

```
{
    "update" : {
        "expression" : "someExpression"
        "expressionNames" : {
            "#foo" : "foo"
        },
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "condition": {
        "consistentRead" = true
    }
}
```

DeleteItem の場合、retryMapping セクションは次の構造を持ちます。

```
{
    "condition": {
        "consistentRead" = true
    }
}
```

使用する別の処理やキーを指定する方法はありません。 AWS AppSync DynamoDB の関数は、同じオブジェクトに対する同じ処理の再試行のみが可能です。また、condition セクションでは conditionalCheckFailedHandler は指定できません。再試行が失敗した場合、 AWS AppSync DynamoDB 関数は Reject戦略に従います。

以下は、失敗した PutItem リクエストを処理する Lambda 関数の例です。ビジネスロジックは呼び出し元を調べます。呼び出し元が jeffTheAdmin の場合は、リクエストを再試行して、現在 DynamoDB にある項目の version と expectedVersion を更新します。それ以外の場合は、ミューテーションを拒否します。

```
exports.handler = (event, context, callback) => {
    console.log("Event: "+ JSON.stringify(event));

    // Business logic goes here.

    var response;
    if ( event.identity.user == "jeffTheAdmin" ) {
        response = {
            "action" : "retry",
            "retryMapping" : {
                "attributeValues" : event.requestMapping.attributeValues,
                "condition" : {
                    "expression" : event.requestMapping.condition.expression,
                    "expressionValues" :
 event.requestMapping.condition.expressionValues
                }
            }
        }
        response.retryMapping.attributeValues.version = { "N" :
 event.currentValue.version.N + 1 }
        response.retryMapping.condition.expressionValues[':expectedVersion'] =
 event.currentValue.version

    } else {
        response = { "action" : "reject" }
    }

    console.log("Response: "+ JSON.stringify(response))
    callback(null, response)
};
```

トランザクション条件式

トランザクション条件式は、TransactWriteItems の 4 つのタイプのオペレーション (PutItem、DeleteItem、UpdateItem、ConditionCheck) すべてに対応するリクエストで使用できます。

PutItem、DeleteItem、および UpdateItem の場合、トランザクション条件式はオプションです。ConditionCheck の場合、トランザクション条件式が必要です。

例 1

次のトランザクション DeleteItem 関数リクエストハンドラーには条件式がありません。その結果、DynamoDB の項目が削除されます。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { postId } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'DeleteItem',
        key: util.dynamodb.toMapValues({ postId }),
      }
    ],
  };
}
```

例 2

次のトランザクション DeleteItem 関数リクエストハンドラーには、その投稿の作成者が特定の名前に等しい場合にのみオペレーションが成功できるトランザクション条件式があります。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { postId, authorName} = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'DeleteItem',
        key: util.dynamodb.toMapValues({ postId }),
        condition: util.transform.toDynamoDBConditionExpression({
          authorName: { eq: authorName },
        }),
      }
    ],
```

```
  };
}
```

条件チェックが失敗すると、TransactionCanceledException が発生し、エラーの詳細が ctx.result.cancellationReasons で返されます。デフォルトでは、その条件チェックの失敗となった DynamoDB の古い項目は ctx.result.cancellationReasons で返されます。

条件を指定する

PutItem、UpdateItem、および DeleteItem の各リクエストオブジェクトはすべて、オプションで condition セクションが指定できます。省略した場合、条件チェックは実行されません。指定した場合、処理が成功するには、条件が true となる必要があります。ConditionCheck では、condition セクションを指定する必要があります。トランザクション全体が成功するためには、条件が true でなければなりません。

condition セクションは以下の構造を持ちます。

```
type TransactConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string };
  expressionValues?: { [key: string]: string };
  returnValuesOnConditionCheckFailure: boolean;
};
```

以下のフィールドに条件を指定します。

expression

更新式そのものを指定します。条件式の記述方法の詳細については、DynamoDB ConditionExpressions のドキュメントを参照してください。このフィールドの指定は必須です。

expressionNames

式の属性名のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される名前のプレースホルダーに対応し、値は DynamoDB の項目の属性名と一致する文字列でなければなりません。このフィールドはオプションであり、expression で使用される式の属性名のプレースホルダーのみを入力します。

expressionValues

式の属性値のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される値のプレースホルダーに対応し、値は型付き値でなければなりません。

「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この指定は必須です。このフィールドはオプションであり、expression で使用される式の属性値のプレースホルダーのみを入力します。

returnValuesOnConditionCheckFailure

条件チェックが失敗した場合に、DynamoDB の項目を取得し直すかどうかを指定します。取得された項目は `ctx.result.cancellationReasons[<index>].item` にあります。ここで `<index>` は、条件チェックに失敗したリクエスト項目のインデックスです。この値のデフォルト値は true です。

計画

`GetItem`、`Scan`、`Query`、`BatchGetItem` および `TransactGetItems` オペレーションを使用して DynamoDB のオブジェクトを読み取る場合、必要な属性を識別するプロジェクションをオプションで指定できます。projection プロパティの構造は以下のとおりです。これらはフィルタに似ています。

```
type DynamoDBExpression = {
  expression: string;
  expressionNames?: { [key: string]: string}
};
```

各フィールドの定義は以下のようになります。

expression

プロジェクション式、これは文字列です。1 つの属性を取得するには、名前を指定します。複数の属性の場合、名前をカンマで区切る必要があります。プロジェクション式の記述の詳細については、「DynamoDB プロジェクション式」のドキュメントを参照してください。このフィールドは必須です。

expressionNames

式の属性名のプレースホルダーを示します。キー - 値のペアの形式になります。キーは、`expression` で使用される名前のプレースホルダーに対応します。値は、DynamoDB の項目の属性名に対応する文字列である必要があります。このフィールドはオプションであり、`expression` で使用される式の属性名のプレースホルダーのみを入力します。`expressionNames` の詳細については、「Amazon DynamoDB のドキュメント」を参照してください。

例 1

次の例は、DynamoDB から属性 `author` と `id` だけが返される JavaScript 関数のプロジェクションセクションです。

```
projection : {
    expression : "#author, id",
    expressionNames : {
        "#author" : "author"
    }
}
```

> **Tip**
>
> SelectionSetList を使用して GraphQL リクエストセレクションセットにアクセスできます。このフィールドでは、要件に応じてプロジェクション式を動的にフレーミングできます。

> **Note**
>
> `Query` と `Scan` のオペレーションでプロジェクション式を使用する場合、`select` の値は `SPECIFIC_ATTRIBUTES` でなければなりません。詳細については、「DynamoDB のドキュメント」を参照してください。

OpenSearch のAWS AppSync JavaScript リゾルバー関数リファレンス

Amazon OpenSearch Service の AWS AppSync リゾルバーを使用すると、GraphQL を使用して、アカウントの既存の OpenSearch Service ドメインにデータを保存および取得できます。リゾルバーにより、受信した GraphQL リクエストを OpenSearch Service リクエストにマッピングし、その後 OpenSearch Service のレスポンスを GraphQL にマッピングすることができます。このセクションでは、サポートされている OpenSearch Service オペレーションの関数リクエストハンドラーとレスポンスハンドラーについて説明します。

リクエスト

ほとんどの OpenSearch Service リクエストオブジェクトは構造が共通しており、変更されるのはほんのわずかです。次の例では、OpenSearch Service ドメインに対して検索を実行します。ここでは、ドキュメントは `post` タイプで `id` にインデックスが作成されます。検索パラメータは `body` セクションで定義され、`query` フィールドで定義されている多くの一般的なクエリの句を使用します。この例では `"Nadia"`、または `"Bailey"`、あるいはその両方を、ドキュメントの `author` フィールドに含むドキュメントを検索します。

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: {
        from: 0,
        size: 50,
        query: {
          bool: {
            should: [
              { match: { author: 'Nadia' } },
              { match: { author: 'Bailey' } },
            ],
          },
        },
      },
    },
  };
}
```

レスポンス

他のデータソースと同様に、OpenSearch Service は GraphQL に変換する必要があるレスポンスを AWS AppSync に送信します。

ほとんどの GraphQL クエリは OpenSearch Service レスポンスから `_source` フィールドを探しています。個別のドキュメントまたはドキュメントのリストを返す検索を行うことができるので、OpenSearch Service で使用される、2 つの共通レスポンスマッピングテンプレートがあります。

結果のリスト

```
export function response(ctx) {
  const entries = [];
  for (const entry of ctx.result.hits.hits) {
    entries.push(entry['_source']);
  }
  return entries;
}
```

個別項目

```
export function response(ctx) {
  return ctx.result['_source']
}
```

`operation` フィールド

> **Note**
>
> これは、リクエストハンドラーにのみ適用されます。

AWS AppSync が OpenSearch Service ドメインに送信する HTTP メソッドまたは動詞 (GET、POST、PUT、HEAD、DELETE)。キーと値の両方が文字列である必要があります。

```
"operation" : "PUT"
```

`path` フィールド

> **Note**
>
> これは、リクエストハンドラーにのみ適用されます。

AWS AppSync からの OpenSearch Service リクエストの検索パス。これはオペレーションの HTTP 動作に対する URL を作成します。キーと値の両方が文字列である必要があります。

```
"path" : "/indexname/type"
```

```
"path" : "/indexname/type/_search"
```

マッピングテンプレートが評価されると、このパスは、OpenSearch Service ドメインが含まれる HTTP リクエストの一部として送信されます。たとえば、前の例では次のように変換される可能性があります。

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

params フィールド

> **Note**
>
> これは、リクエストハンドラーにのみ適用されます。

検索実行時のアクションを指定するために使用され、一般に、query 値を body 内に設定します。ただし、レスポンスのフォーマットなど、他のいくつかの機能を設定できます。

- ヘッダ

  ヘッダー情報は、キーと値のペアです。キーと値の両方が文字列である必要があります。以下に例を示します。

  ```
  "headers" : {
      "Content-Type" : "application/json"
  }
  ```

  > **Note**
  >
  > AWS AppSync は現在、 として JSON のみをサポートしていますContent-Type。

- queryString

  一般的なオプション (JSON レスポンスのコードフォーマットなど) を指定するキーと値のペア。キーと値の両方が文字列である必要があります。たとえば、整形表示の JSON を取得する場合は、次を使用します。

  ```
  "queryString" : {
  ```

```
    "pretty" : "true"
}
```

- body

  これはリクエストの主要部分であり、 AWS AppSync が OpenSearch Service ドメインに対して適切な形式の検索リクエストを作成できるようにします。キーはオブジェクトで構成される文字列である必要があります。2 つのデモを以下に示します。

例 1

都市が「seattle」に一致するすべてのドキュメントを返します。

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: { from: 0, size: 50, query: { match: { city: 'seattle' } } },
    },
  };
}
```

例 2

都市または州が「washington」に一致するすべてのドキュメントを返します。

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: {
        from: 0,
        size: 50,
        query: {
          multi_match: { query: 'washington', fields: ['city', 'state'] },
        },
```

```
      },
    },
  };
}
```

渡す変数

> **Note**
>
> これは、リクエストハンドラーにのみ適用されます。

リクエストハンドラーの評価の一部として変数を渡すこともできます。たとえば、次のような GraphQL クエリがあるとします。

```
query {
    searchForState(state: "washington"){
        ...
    }
}
```

関数リクエストハンドラーは次のようになります。

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: {
        from: 0,
        size: 50,
        query: {
          multi_match: { query: ctx.args.state, fields: ['city', 'state'] },
        },
      },
    },
  };
}
```

Lambda のAWS AppSync JavaScript リゾルバー関数リファレンス

AWS AppSync 関数とリゾルバーを使用して、 アカウントにある Lambda 関数を呼び出すことができます。クライアントに返す前に、リクエストペイロードと Lambda 関数からのレスポンスを調整することができます。実行する操作のタイプをリクエストオブジェクトで指定することもできます。このセクションでは、サポートされる Lambda 操作に対するリクエストについて説明します。

オブジェクトをリクエストする

Lambda リクエストオブジェクトは、Lambda 関数に関連するフィールドを処理します。

```
export type LambdaRequest = {
  operation: 'Invoke' | 'BatchInvoke';
  invocationType?: 'RequestResponse' | 'Event';
  payload: unknown;
};
```

以下は、ペイロードデータが GraphQL スキーマの `getPost` フィールドであり、コンテキストの引数を含む `invoke` オペレーションを使用する例です。

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

マッピングドキュメント全体は、使用する Lambda 関数に入力として渡されます。これにより、先ほどのサンプルは以下のようになります。

```
{
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "input": {
        "id": "postId1",
      }
    }
  }
```

```
}
```

Operation

Lambda データソースでは、operation フィールドで 2 つのオペレーションである Invoke と BatchInvoke を定義できます。Invoke オペレーションは、GraphQL フィールドリゾルバーごとに Lambda 関数を呼び出すことを AWS AppSync に知らせます。 BatchInvoke は、現在の GraphQL フィールドのリクエストをバッチ処理するように AWS AppSync に指示します。operation フィールドは必須です。

Invoke では、解決されたリクエストは Lambda 関数の入力ペイロードと一致します。上記の例を変更してみましょう。

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

これは解決され、Lambda 関数に渡されます。この関数は次のようになります。

```
{
  "operation": "Invoke",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

BatchInvoke では、リクエストはバッチ内のすべてのフィールドリゾルバーに適用されます。簡潔にするために、 AWS AppSync はすべてのリクエストpayload値を、リクエストオブジェクトに一致する 1 つのオブジェクトのリストにマージします。以下のリクエストハンドラーの例ではマージを示しています。

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: ctx,
```

```
  };
}
```

このリクエストは評価され、次のマッピングドキュメントに解決されます。

```
{
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

payload リストの各要素は、1 回のバッチ項目に対応しています。また、Lambda 関数はリスト形式のレスポンスを返すことが予期され、次のように、各項目はリクエストで送信された順序になります。

```
[
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
 1
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
 2
  { "data": {...}, "errorMessage": null, "errorType": null }  // result for batch item
 3
]
```

ペイロード

payload フィールドは、任意のデータを Lambda 関数に渡す際に使用されるコンテナです。operation フィールドが BatchInvoke に設定されている場合、 AWS AppSync は既存の payload 値をリストにまとめます。payload フィールドはオプションです。

呼び出しタイプ

Lambda データソースでは、2 つの呼び出しタイプ RequestResponse と Event を定義できます。呼び出しタイプは、Lambda API で定義されている呼び出しタイプと同義です。RequestResponse 呼び出しタイプ lets AWS AppSync は Lambda 関数を同期的に呼び出してレスポンスを待ちます。Event 呼び出しにより、Lambda 関数を非同期的に呼び出すことができます。Lambda が Event 呼び出しタイプのリクエストを処理する方法の詳細については、「非同期呼び出し」を参照

してください。invocationType フィールドはオプションです。このフィールドがリクエストに含まれていない場合、 AWS AppSync はデフォルトで RequestResponse 呼び出しタイプになります。

invocationType フィールドでは、解決されたリクエストは Lambda 関数の入力ペイロードと一致します。上記の例を変更してみましょう。

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    invocationType: 'Event',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

これは解決され、Lambda 関数に渡されます。この関数は次のようになります。

```
{
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

BatchInvoke オペレーションをEvent呼び出しタイプフィールドと組み合わせて使用すると、AWS AppSync は上記の同じ方法でフィールドリゾルバーをマージし、リクエストは値payloadのリストである を使用して非同期イベントとして Lambda 関数に渡されます。Event 呼び出しタイプのリクエストからのレスポンスは、レスポンスハンドラーのない null 値になります。

```
{
  "data": {
    "field": null
  }
}
```

Event 呼び出しタイプのリゾルバーのリゾルバーキャッシュは、キャッシュヒットがあった場合に Lambda に送信されないため、無効にすることをお勧めします。

レスポンスオブジェクト

他のデータソースと同様に、Lambda 関数は GraphQL タイプに変換する必要があるレスポンスを AWS AppSync に送信します。Lambda 関数の結果は、`context` result プロパティに含まれます (`context.result`)。

Lambda 関数のレスポンスの形状と GraphQL タイプの形状が正確に一致する場合は、以下の関数レスポンスハンドラーを使用して、レスポンスを転送できます。

```
export function response(ctx) {
  return ctx.result
}
```

レスポンスオブジェクトに適用される形状の制限や必須フィールドはありません。ただし、GraphQL が厳密に型指定されているので、解決されたレスポンスは予想される GraphQL タイプに一致する必要があります。

Lambda 関数のバッチ処理されたレスポンス

`operation` フィールドが `BatchInvoke` に設定されている場合、 AWS AppSync は Lambda 関数から返される項目のリストを想定します。 AWS AppSync が各結果を元のリクエスト項目にマッピングするには、レスポンスリストのサイズと順序が一致している必要があります。レスポンスリストに `null` 項目がある場合も有効です。その場合、`ctx.result` が null に設定されます。

EventBridge データソースのAWS AppSync JavaScript リゾルバー関数リファレンス

EventBridge データソースで使用される AWS AppSync リゾルバー関数のリクエストとレスポンスにより、Amazon EventBridge バスにカスタムイベントを送信できます。

リクエスト

リクエストハンドラーを使用すると、複数のカスタムイベントを EventBridge イベントバスに送信できます。

```
export function request(ctx) {
  return {
    "operation" : "PutEvents",
    "events" : [{}]
```

```
  }
}
```

EventBridge PutEvents リクエストには以下のタイプ定義があります。

```
type PutEventsRequest = {
  operation: 'PutEvents'
  events: {
    source: string
    detail: { [key: string]: any }
    detailType: string
    resources?: string[]
    time?: string // RFC3339 Timestamp format
  }[]
}
```

レスポンス

PutEvents 操作が成功すると、EventBridge からのレスポンスが ctx.result に含まれます。

```
export function response(ctx) {
  if(ctx.error)
    util.error(ctx.error.message, ctx.error.type, ctx.result)
  else
    return ctx.result
}
```

InternalExceptions や Timeouts などの PutEvents 操作の実行中に発生したエラーは、ctx.error に表示されます。EventBridge の一般的なエラーのリストについては、EventBridge の一般的なエラーのリファレンスを参照してください。

result には以下のタイプ定義があります。

```
type PutEventsResult = {
  Entries: {
    ErrorCode: string
    ErrorMessage: string
    EventId: string
  }[]
  FailedEntryCount: number
}
```

- エントリ

  取り込まれたイベントは、成功と失敗の両方の結果になります。取り込みが成功すると、エントリには EventID が含まれます。それ以外の場合は、ErrorCode と ErrorMessage を使用してエントリの問題を特定できます。

  各レコードの応答要素のインデックスは、リクエスト配列のインデックスと同じです。

- FailedEntryCount

  失敗したエントリの数。この値は整数として表されます。

PutEvents のレスポンスの詳細については、「PutEvents」を参照してください。

サンプルレスポンスの例: 1

次の例は、2 つのイベントが成功する PutEvents 操作です。

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ],
    "FailedEntryCount" : 0
}
```

サンプルレスポンスの例: 2

次の例は、3 つのイベント (2 つの成功、1 つの失敗) がある PutEvents 操作です。

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        },
        {
```

```
            "ErrorCode" : "SampleErrorCode",
            "ErrorMessage" : "Sample Error Message"
        }
    ],
    "FailedEntryCount" : 1
}
```

PutEvents フィールド

PutEvents には、次のマッピングテンプレートフィールドが含まれています。

- バージョン

  すべてのリクエストマッピングテンプレートに共通で、version フィールドはテンプレートが使用するバージョンを定義します。このフィールドは必須です。値 2018-05-29 は、EventBridge マッピングテンプレートでサポートされている唯一のバージョンです。
- 操作

  サポートされている操作は PutEvents のみです。この操作により、カスタムイベントをイベントバスに追加できます。
- イベント

  イベントバスに追加されるイベントの配列。この配列には 1 ~ 10 個の項目が割り当てられている必要があります。

  Event オブジェクトには以下のフィールドがあります。
  - "source": イベントのソースを識別する文字列。
  - "detail": イベントに関する情報をアタッチするのに使用できる JSON オブジェクト。このフィールドは空のマップ ({ }) でもかまいません。
  - "detailType: イベントの種類を識別する文字列。
  - "resources": イベントに関わるリソースを識別する文字列の JSON 配列 このフィールドは、空白の配列でもかまいません。
  - "time": 文字列として提供されるイベントのタイムスタンプ。これは RFC3339 タイムスタンプ形式に従う必要があります。

以下のスニペットは有効な Event オブジェクトの例です。

例 1

```
{
    "source" : "source1",
    "detail" : {
        "key1" : [1,2,3,4],
        "key2" : "strval"
    },
    "detailType" : "sampleDetailType",
    "resources" : ["Resouce1", "Resource2"],
    "time" : "2022-01-10T05:00:10Z"
}
```

例 2

```
{
    "source" : "source1",
    "detail" : {},
    "detailType" : "sampleDetailType"
}
```

例 3

```
{
    "source" : "source1",
    "detail" : {
        "key1" : 1200
    },
    "detailType" : "sampleDetailType",
    "resources" : []
}
```

None データソースのAWS AppSync JavaScript リゾルバー関数リファレンス

データソースタイプ None の AWS AppSync リゾルバー関数のリクエストとレスポンスを使用すると、 AWS AppSync ローカルオペレーションのリクエストをシェイプできます。

リクエスト

リクエストハンドラーはシンプルで、payload フィールド経由で可能な限り多くのコンテキスト情報を渡すことができます。

```
type NONERequest = {
  payload: any;
};
```

以下に示しているのは、ペイロードにフィールド引数を渡すようにした例です。

```
export function request(ctx) {
  return {
    payload: context.args
  };
}
```

payload フィールドの値は関数応答ハンドラーに転送され、context.result で使用できるようになります。

ペイロード

payload フィールドは任意のデータを渡すためのコンテナで、そのデータを関数レスポンスハンドラーに渡すことができます。

payload フィールドはオプションです。

レスポンス

データソースがないため、payload フィールドの値が関数レスポンスハンドラーに転送され、context.result プロパティに設定されます。

payload フィールド値の形状と GraphQL タイプの形状が正確に一致する場合、以下のレスポンスハンドラーを使用して、レスポンスを転送できます。

```
export function response(ctx) {
  return ctx.result;
}
```

返されたレスポンスに適用される形状の制限や必須フィールドはありません。ただし、GraphQL が厳密に型指定されているので、解決されたレスポンスは予想される GraphQL タイプに一致する必要があります。

HTTP のAWS AppSync JavaScript リゾルバー関数リファレンス

AWS AppSync HTTP リゾルバー関数を使用すると、 AWS AppSync から任意の HTTP エンドポイントにリクエストを送信し、HTTP エンドポイントからのレスポンスを AWS AppSync に返すことができます。リクエストハンドラーを使用して、呼び出されるオペレーションの特性に関して、 AWS AppSync にヒントを渡すこともできます。このセクションでは、サポートされる HTTP リゾルバーの異なる設定について説明します。

リクエスト

```
type HTTPRequest = {
  method: 'PUT' | 'POST' | 'GET' | 'DELETE' | 'PATCH';
  params?: {
    query?: { [key: string]: any };
    headers?: { [key: string]: string };
    body?: any;
  };
  resourcePath: string;
};
```

以下のスニペットは、text/plain 本文のある HTTP POST リクエストの例です。

```
export function request(ctx) {
  return {
    method: 'POST',
    params: {
      headers: { 'Content-Type': 'text/plain' },
      body: 'this is an example of text body',
    },
    resourcePath: '/',
  };
}
```

[メソッド]

> **Note**
>
> これは、リクエストハンドラーにのみ適用されます。

AWS AppSync が HTTP エンドポイントに送信する HTTP メソッドまたは動詞 (GET、POST、PUT、PATCH、または DELETE)。

```
"method": "PUT"
```

ResourcePath

> **Note**
>
> これは、リクエストハンドラーにのみ適用されます。

アクセス対象のリソースパスです。リソースパスは、HTTP データソースのエンドポイントとともに、 AWS AppSync サービスがリクエストを行う URL を形成します。

```
"resourcePath": "/v1/users"
```

リクエストが評価されると、このパスは、HTTP エンドポイントが含まれる HTTP リクエストの一部として送信されます。たとえば、前の例では次のように変換される可能性があります。

```
PUT <endpoint>/v1/users
```

Params フィールド

> **Note**
>
> これは、リクエストハンドラーにのみ適用されます。

検索実行時のアクションを指定するために使用され、一般に、query 値を body 内に設定します。ただし、レスポンスのフォーマットなど、他のいくつかの機能を設定できます。

ヘッダ

ヘッダー情報は、キーと値のペアです。キーと値の両方が文字列である必要があります。

例:

```
"headers" : {
```

```
    "Content-Type" : "application/json"
}
```

現在サポートされている Content-Type ヘッダーは以下のとおりです。

```
text/*
application/xml
application/json
application/soap+xml
application/x-amz-json-1.0
application/x-amz-json-1.1
application/vnd.api+json
application/x-ndjson
```

以下の HTTP ヘッダーを設定することはできません。

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

一般的なオプション (JSON レスポンスのコードフォーマットなど) を指定するキーと値のペア。キーと値の両方が文字列である必要があります。次の例では、?type=json としてクエリ文字列を送信する方法を示しています。

```
"query" : {
    "type" : "json"
}
```

body

ボディには、設定の際に選択する HTTP リクエストボディが含まれています。リクエストボディは、コンテンツタイプが charset に指定されている場合を除き、常に UTF-8 でエンコードされた文字列です。

```
"body":"body string"
```

レスポンス

例を参照してください。こちらへ。

Amazon RDS のAWS AppSync JavaScript リゾルバー関数リファレンス

AWS AppSync RDS 関数とリゾルバーを使用すると、デベロッパーは RDS Data API を使用して Amazon Aurora クラスターデータベースにSQLクエリを送信し、これらのクエリの結果を取得できます。 AWS AppSyncの`rds`モジュール`sql`タグ付きテンプレートを使用するか、`rds`モジュールの`select`、`insert`、、および `remove`ヘルパー関数を使用して`update`、Data API に送信されるSQLステートメントを記述できます。 は、RDS Data Service の `ExecuteStatement`アクション AWS AppSync を利用して、データベースに対して SQL ステートメントを実行します。

トピック

- SQL タグ付きテンプレート
- ステートメントの作成
- データの取得
- ユーティリティ関数
- SQL SELECT
- SQL INSERT
- SQL UPDATE
- SQL DELETE
- キャスティング

SQL タグ付きテンプレート

AWS AppSyncの`sql`タグ付けされたテンプレートを使用すると、テンプレート式を使用してランタイムに動的値を受信できる静的ステートメントを作成できます。 は式値から変数マップ AWS AppSync を構築して、Amazon Aurora Serverless Data API に送信される`SqlParameterized`クエリを構築します。この方法では、実行時に渡される動的な値によって元のステートメントを変更することができないため、意図しない実行が発生する可能性があります。動的な値はすべてパラメータとして渡され、元のステートメントを変更することはできず、データベースによって実行されることもありません。これにより、クエリが SQL インジェクション攻撃を受けにくくなります。

> **Note**
>
> いずれの場合も、SQL ステートメントを作成するときは、セキュリティガイドラインに従って、入力として受け取ったデータを適切に処理する必要があります。

> **Note**
>
> `sql` タグ付きテンプレートは変数値の受け渡しのみをサポートします。式を使用して列名やテーブル名を動的に指定することはできません。ただし、ユーティリティ関数を使用して動的ステートメントを作成することはできます。

次の例では、実行時に GraphQL クエリで動的に設定される `col` 引数の値に基づいてフィルタリングするクエリを作成します。この値は、タグ式を使用してのみステートメントに追加できます。

```
import { sql, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const query = sql`
SELECT * FROM table
WHERE column = ${ctx.args.col}`
  ;
  return createMySQLStatement(query);
}
```

すべての動的な値を変数マップに渡すことで、データベースエンジンに依存して値を安全に処理し、サニタイズします。

ステートメントの作成

関数とリゾルバーは、MySQL データベースと PostgreSQL データベースとやり取りできます。`createMySQLStatement` と `createPgStatement` をそれぞれ使用してステートメントを作成します。例えば、`createMySQLStatement` は MySQL クエリを作成できます。これらの関数は最大 2 つのステートメントを受け付けるので、リクエストですぐに結果を取得する必要がある場合に便利です。MySQL では、以下のことが可能です。

```
import { sql, createMySQLStatement } from '@aws-appsync/utils/rds';
```

```
export function request(ctx) {
    const { id, text } = ctx.args;
    const s1 = sql`insert into Post(id, text) values(${id}, ${text})`;
    const s2 = sql`select * from Post where id = ${id}`;
    return createMySQLStatement(s1, s2);
}
```

> **Note**
>
> createPgStatement および createMySQLStatement は、sql タグ付きテンプレートで作成されたステートメントをエスケープしたり、引用したりすることはありません。

データの取得

実行された SQL ステートメントの結果は、context.result オブジェクトのレスポンスハンドラで参照できます。結果は、ExecuteStatement アクションからのレスポンス要素を含む JSON 文字列です。解析された結果は次のとおりです。

```
type SQLStatementResults = {
    sqlStatementResults: {
        records: any[];
        columnMetadata: any[];
        numberOfRecordsUpdated: number;
        generatedFields?: any[]
    }[]
}
```

toJsonObject ユーティリティを使用して、返された行を表す JSON オブジェクトのリストに結果を変換できます。以下に例を示します。

```
import { toJsonObject } from '@aws-appsync/utils/rds';

export function response(ctx) {
    const { error, result } = ctx;
    if (error) {
        return util.appendError(
            error.message,
            error.type,
            result
```

```
        )
    }
    return toJsonObject(result)[1][0]
}
```

toJsonObject はステートメントの結果の配列を返すことに注意してください。1 つのステートメントを指定した場合、配列の長さは 1 です。2 つのステートメントを指定した場合、配列の長さは 2 です。配列内の各結果には、0 以上の行が含まれます。結果の値が無効な場合や予期しないものである場合、toJsonObject は null を返します。

ユーティリティ関数

AWS AppSync RDS モジュールのユーティリティヘルパーを使用して、データベースとやり取りできます。

SQL SELECT

select ユーティリティは、リレーショナルデータベースにクエリを実行する SELECT ステートメントを作成します。

基本的な使用法

基本的な形式では、クエリを実行するテーブルを指定できます。

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // "SELECT * FROM "persons"
    return createPgStatement(select({table: 'persons'}));
}
```

なお、テーブル識別子でスキーマを指定することもできます。

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // SELECT * FROM "private"."persons"
```

```
    return createPgStatement(select({table: 'private.persons'}));
}
```

列の指定

columns プロパティで列を指定できます。これを値に設定しない場合、デフォルトでは * になります。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name']
    }));
}
```

また、列のテーブルも指定できます。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "persons"."name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'persons.name']
    }));
}
```

LIMIT と OFFSET

limit と offset をクエリに適用できます。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // LIMIT :limit
```

```
    // OFFSET :offset
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        limit: 10,
        offset: 40
    }));
}
```

ORDER BY

結果は orderBy プロパティを使用してソートできます。列とオプションの dir プロパティを指定するオブジェクトの配列を指定します。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name" FROM "persons"
    // ORDER BY "name", "id" DESC
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        orderBy: [{column: 'name'}, {column: 'id', dir: 'DESC'}]
    }));
}
```

フィルター

特殊条件オブジェクトを使用してフィルターを構築できます。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}}
    }));
}
```

以下のフィルターを組み合わせることもできます。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME and "id" > :ID
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}, id: {gt: 10}}
    }));
}
```

また、OR ステートメントも作成できます。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME OR "id" > :ID
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: { or: [
            { name: { eq: 'Stephane'} },
            { id: { gt: 10 } }
        ]}
    }));
}
```

not を使用して条件を無効にすることもできます。

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE NOT ("name" = :NAME AND "id" > :ID)
    return createPgStatement(select({
        table: 'persons',
```

```
        columns: ['id', 'name'],
        where: { not: [
            { name: { eq: 'Stephane'} },
            { id: { gt: 10 } }
        ]}
    }));
}
```

以下の演算子を使用して、値を比較することもできます。

演算子	説明	可能な値型
eq	Equal	number, string, boolean
ne	Not equal	number, string, boolean
le	Less than or equal	number, string
lt	Less than	number, string
ge	Greater than or equal	number, string
gt	Greater than	number, string
contains	Like	string
notContains	Not like	string
beginsWith	Starts with prefix	string
between	Between two values	number, string
attributeExists	The attribute is not null	number, string, boolean
size	checks the length of the element	string

SQL INSERT

`insert` ユーティリティによって、INSERT オペレーションを使用してデータベースに単一行の項目を簡単に挿入できます。

単一項目の挿入

項目を挿入するには、テーブルを指定して、値のオブジェクトを渡します。オブジェクトキーはテーブルの列にマッピングされます。列名は自動的にエスケープされ、値は変数マップを使用してデータベースに送信されます。

```
import { insert, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({ table: 'persons', values });

    // Generates statement:
    // INSERT INTO `persons`(`name`)
    // VALUES(:NAME)
    return createMySQLStatement(insertStatement)
}
```

MySQL ユースケース

insert の後に select を組み合わせて、挿入した行を取得できます。

```
import { insert, select, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({  table: 'persons', values });
    const selectStatement = select({
        table: 'persons',
        columns: '*',
        where: { id: { eq: values.id } },
        limit: 1,
    });

    // Generates statement:
    // INSERT INTO `persons`(`name`)
    // VALUES(:NAME)
    // and
    // SELECT *
    // FROM `persons`
    // WHERE `id` = :ID
    return createMySQLStatement(insertStatement, selectStatement)
```

```
}
```

Postgres のユースケース

Postgres では、[returning](#) を使用して、挿入した行からデータを取得できます。* または列名の配列が受け入れられます。

```
import { insert, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({
        table: 'persons',
        values,
        returning: '*'
    });

    // Generates statement:
    // INSERT INTO "persons"("name")
    // VALUES(:NAME)
    // RETURNING *
    return createPgStatement(insertStatement)
}
```

SQL UPDATE

update ユーティリティでは、既存の行を更新できます。条件オブジェクトを使用すると、条件を満たすすべての行の指定された列に変更を適用できます。例えば、このミューテーションを可能にするスキーマがあるとします。Person の name を 3 の id 値で更新したいのですが、これは 2000 年以降にそれら (known_since) がわかっている場合に限ります。

```
mutation Update {
    updatePerson(
        input: {id: 3, name: "Jon"},
        condition: {known_since: {ge: "2000"}}
    ) {
    id
    name
  }
}
```

更新リゾルバーは以下のようになります。

```
import { update, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: { id, ...values }, condition } = ctx.args;
    const where = {
        ...condition,
        id: { eq: id },
    };
    const updateStatement = update({
        table: 'persons',
        values,
        where,
        returning: ['id', 'name'],
    });

    // Generates statement:
    // UPDATE "persons"
    // SET "name" = :NAME, "birthday" = :BDAY, "country" = :COUNTRY
    // WHERE "id" = :ID
    // RETURNING "id", "name"
    return createPgStatement(updateStatement)
}
```

条件にチェックを追加して、3 と等しいプライマリキー id を持つ行だけが更新されるようにすることができます。同様に、Postgres inserts の場合も、returning を使用して変更されたデータを返すことができます。

SQL DELETE

remove ユーティリティでは、既存の行を削除できます。条件オブジェクトは、条件を満たすすべての行で使用できます。なお、delete はJavaScriptの予約キーワードです。代わりに remove を使ってください。

```
import { remove, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: { id }, condition } = ctx.args;
    const where = { ...condition, id: { eq: id } };
    const deleteStatement = remove({
        table: 'persons',
        where,
        returning: ['id', 'name'],
```

```
    });

    // Generates statement:
    // DELETE "persons"
    // WHERE "id" = :ID
    // RETURNING "id", "name"
    return createPgStatement(updateStatement)
}
```

キャスティング

ステートメントで使用する正しいオブジェクト型に関して、さらに特異度が必要となる場合もあるでしょう。指定された型ヒントを使用して、パラメータのタイプを指定できます。 は、Data API と同じ型ヒント AWS AppSync をサポートしています。モジュールの typeHint関数を使用してパラメータを AWS AppSync rdsキャストできます。

次の例では、JSON オブジェクトとしてキャストされた値として配列を送信できます。-> 演算子を使用して JSON 配列内にある index 2 の要素を取得します。

```
import { sql, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/
rds';

export function request(ctx) {
    const arr = ctx.args.list_of_ids
    const statement = sql`select ${typeHint.JSON(arr)}->2 as value`
    return createPgStatement(statement)
}

export function response(ctx) {
    return toJsonObject(ctx.result)[0][0].value
}
```

キャスティングは、DATE、TIME、TIMESTAMP の処理や比較を行うときにも役立ちます。

```
import { select, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const when = ctx.args.when
    const statement = select({
        table: 'persons',
        where: { createdAt : { gt: typeHint.DATETIME(when) } }
```

```
    })
    return createPgStatement(statement)
}
```

現在の日付と時刻を送信する方法について別の例を以下に示します。

```
import { sql, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const now = util.time.nowFormatted('YYYY-MM-dd HH:mm:ss')
    return createPgStatement(sql`select ${typeHint.TIMESTAMP(now)}`)
}
```

使用可能な型ヒント

- typeHint.DATE - 対応するパラメータが、DATE 型のオブジェクトとしてデータベースに送信されます。受け入れられる形式は YYYY-MM-DD です。
- typeHint.DECIMAL - 対応するパラメータが、DECIMAL 型のオブジェクトとしてデータベースに送信されます。
- typeHint.JSON - 対応するパラメータが、JSON 型のオブジェクトとしてデータベースに送信されます。
- typeHint.TIME - 対応する文字列パラメータ値が、TIME 型のオブジェクトとしてデータベースに送信されます。受け入れられる形式は HH:MM:SS[.FFF] です。
- typeHint.TIMESTAMP - 対応する文字列パラメータ値が、TIMESTAMP 型のオブジェクトとしてデータベースに送信されます。受け入れられる形式は YYYY-MM-DD HH:MM:SS[.FFF] です。
- typeHint.UUID - 対応する文字列パラメータ値が、UUID 型のオブジェクトとしてデータベースに送信されます。

Amazon Bedrock ランタイムのAWS AppSync JavaScript リゾルバーと関数リファレンス

AWS AppSync 関数とリゾルバーを使用して、 の Amazon Bedrock でモデルを呼び出すことができます AWS アカウント。リクエストペイロードとモデル呼び出し関数からのレスポンスを、クライアントに返す前に形成できます。Amazon Bedrock ランタイムの InvokeModel API または Converse API を使用できます。このセクションでは、サポートされている Amazon Bedrock オペレーションのリクエストについて説明します。

> **Note**
>
> AWS AppSync は、10 秒以内に完了する同期呼び出しのみをサポートします。Amazon Bedrock のストリーム APIs を呼び出すことはできません。 は AWS AppSync 、API と同じリージョンでの基盤モデルと推論プロファイルの呼び出し AWS AppSync のみをサポートしています。

オブジェクトをリクエストする

`InvokeModel` リクエストオブジェクトを使用すると、Amazon Bedrock の `InvokeModel` API を操作できます。

```
type BedrockInvokeModelRequest = {
  operation: 'InvokeModel';
  modelId: string;
  body: any;
  guardrailIdentifier?: string;
  guardrailVersion?: string;
  guardrailTrace?: string;
}
```

`Converse` リクエストオブジェクトを使用すると、Amazon Bedrock の `Converse` API を操作できます。

```
type BedrockConverseRequest = {
  operation: 'Converse';
  modelId: string;
  messages: BedrockMessage[];
  additionalModelRequestFields?: any;
  additionalModelResponseFieldPaths?: string[];
  guardrailConfig?: BedrockGuardrailConfig;
  inferenceConfig?: BedrockInferenceConfig;
  promptVariables?: { [key: string]: BedrockPromptVariableValues }[];
  system?: BedrockSystemContent[];
  toolConfig?: BedrockToolConfig;
}
```

詳細については、このトピックのタイプリファレンス後半のセクションを参照してください。

関数とリゾルバーから、リクエストオブジェクトを直接構築することも、@aws-appsync/utils/ai のヘルパー関数を使用してリクエストを作成することもできます。リクエストでモデル ID (modelId) を指定するときは、モデル ID またはモデル ARN を使用できます。

次の例では、 `invokeModel`関数を使用して、Amazon Titan Text G1 - Lite (amazon.titan-text-lite-v1) を使用してテキストを要約します。設定されたガードレールは、プロンプトフロー内の不要なコンテンツを識別してブロックまたはフィルタリングするために使用されます。Amazon Bedrock ガードレールの詳細については、「Amazon Bedrock ユーザーガイド」を参照してください。

> **Important**
>
> お客様は、安全なアプリケーション開発と、プロンプトインジェクションなどの脆弱性の防止に責任を負います。詳細については、「Amazon Bedrock ユーザーガイド」の「プロンプトインジェクションのセキュリティ」を参照してください。

```
import { invokeModel } from '@aws-appsync/utils/ai'
export function request(ctx) {
  return invokeModel({
    modelId: 'amazon.titan-text-lite-v1',
    guardrailIdentifier: "zabcd12345678",
    guardrailVersion: "1",
    body: { inputText: `Summarize this text in less than 100 words. : \n<text>
${ctx.stash.text ?? ctx.env.DEFAULT_TEXT}</text>` },
  })
}

export function response(ctx) {
  return ctx.result.results[0].outputText
}
```

次の例では、クロスリージョン推論プロファイル (us.anthropic.claude-3-5-haiku-20241022-v1:0) を持つ `converse`関数を使用します。Amazon Bedrock ユーザーガイドの推論プロファイルの Amazon Bedrock の前提条件の詳細

注意: アプリケーションを安全に開発し、プロンプトインジェクションなどの脆弱性を防ぐ責任があります。

```
import { converse } from '@aws-appsync/utils/ai'
```

```
export function request(ctx) {
  return converse({
    modelId: 'us.anthropic.claude-3-5-haiku-20241022-v1:0',
    system: [
      {
        text: `
You are a database assistant that provides SQL queries to retrieve data based on a
 natural language request.
${ctx.args.explain ? 'Explain your answer' : 'Do not explain your answer'}.
Assume a database with the following tables and columns exists:

Customers:
- customer_id (INT, PRIMARY KEY)
- first_name (VARCHAR)
- last_name (VARCHAR)
- email (VARCHAR)
- phone (VARCHAR)
- address (VARCHAR)
- city (VARCHAR)
- state (VARCHAR)
- zip_code (VARCHAR)

Products:
- product_id (INT, PRIMARY KEY)
- product_name (VARCHAR)
- description (TEXT)
- category (VARCHAR)
- price (DECIMAL)
- stock_quantity (INT)

Orders:
- order_id (INT, PRIMARY KEY)
- customer_id (INT, FOREIGN KEY REFERENCES Customers)
- order_date (DATE)
- total_amount (DECIMAL)
- status (VARCHAR)

Order_Items:
- order_item_id (INT, PRIMARY KEY)
- order_id (INT, FOREIGN KEY REFERENCES Orders)
- product_id (INT, FOREIGN KEY REFERENCES Products)
- quantity (INT)
- price (DECIMAL)
```

```
Reviews:
- review_id (INT, PRIMARY KEY)
- product_id (INT, FOREIGN KEY REFERENCES Products)
- customer_id (INT, FOREIGN KEY REFERENCES Customers)
- rating (INT)
- comment (TEXT)
- review_date (DATE)`,
      },
    ],
    messages: [
      {
        role: 'user',
        content: [{ text: `<request>${ctx.args.text}:</request>` }],
      },
    ],
  })
}

export function response(ctx) {
  return ctx.result.output.message.content[0].text
}
```

次の例ではconverse、 を使用して構造化レスポンスを作成します。DB スキーマリファレンスには環境変数を使用し、攻撃を防ぐためにガードレールを設定することに注意してください。

```
import { converse } from '@aws-appsync/utils/ai'

export function request(ctx) {
  return generateObject({
    modelId: ctx.env.HAIKU3_5, // keep the model in an env variable
    prompt: ctx.args.query,
    shape: objectType(
      {
        sql: stringType('the sql query to execute as a javascript template string.'),
        parameters: objectType({}, 'the placeholder parameters for the query, if
 any.'),
      },
      'the sql query to execute along with the place holder parameters',
    ),
    system: [
      {
        text: `
```

```
You are a database assistant that provides SQL queries to retrieve data based on a
 natural language request.

Assume a database with the following tables and columns exists:

${ctx.env.DB_SCHEMA_CUSTOMERS}
${ctx.env.DB_SCHEMA_ORDERS}
${ctx.env.DB_SCHEMA_ORDER_ITEMS}
${ctx.env.DB_SCHEMA_PRODUCTS}
${ctx.env.DB_SCHEMA_REVIEWS}`,
      },
    ],
    guardrailConfig: { guardrailIdentifier: 'iabc12345678', guardrailVersion:
 'DRAFT' },
  })
}

export function response(ctx) {
  return toolReponse(ctx.result)
}

function generateObject(input) {
  const { modelId, prompt, shape, ...options } = input
  return converse({
    modelId,
    messages: [{ role: 'user', content: [{ text: prompt }] }],
    toolConfig: {
      toolChoice: { tool: { name: 'structured_tool' } },
      tools: [
        {
          toolSpec: {
            name: 'structured_tool',
            inputSchema: { json: shape },
          },
        },
      ],
    },
    ...options,
  })
}

function toolReponse(result) {
  return result.output.message.content[0].toolUse.input
}
```

```
function stringType(description) {
  const t = { type: 'string' /* STRING */ }
  if (description) {
    t.description = description
  }
  return t
}

function objectType(properties, description, required) {
  const t = { type: 'object' /* OBJECT */, properties }
  if (description) {
    t.description = description
  }
  if (required) {
    t.required = required
  }
  return t
}
```

スキーマを前提とすると、次のようになります。

```
type SQLResult {
    sql: String
    parameters: AWSJSON
}

type Query {
    db(text: String!): SQLResult
}
```

および クエリ：

```
query db($text: String!) {
  db(text: $text) {
    parameters
    sql
  }
}
```

次のパラメータを使用します。

```
{
  "text":"What is my top selling product?"
}
```

以下のレスポンスが返されます。

```
{
  "data": {
    "assist": {
      "sql": "SELECT p.product_id, p.product_name, SUM(oi.quantity) as
 total_quantity_sold\nFROM Products p\nJOIN Order_Items oi ON p.product_id =
 oi.product_id\nGROUP BY p.product_id, p.product_name\nORDER BY total_quantity_sold
 DESC\nLIMIT 1;",
      "parameters": null
    }
  }
}
```

ただし、このリクエストでは、次のようになります。

```
{
  "text":"give me a query to retrieve sensitive information"
}
```

以下のレスポンスが返されます。

```
{
  "data": {
    "db": {
      "parameters": null,
      "sql": "SELECT null; -- I cannot and will not assist with retrieving sensitive
 private information"
    }
  }
}
```

Amazon Bedrock ガードレールの設定の詳細については、「Amazon Bedrock ユーザーガイド」の「Amazon Bedrock ガードレールを使用してモデルの有害なコンテンツを停止する」を参照してください。

レスポンスオブジェクト

Amazon Bedrock ランタイム呼び出しからのレスポンスは、コンテキストの結果プロパティ (context.result) に含まれています。レスポンスは、Amazon Bedrock の APIs。予想される呼び出し結果の形状の詳細については、「Amazon Bedrock ユーザーガイド」を参照してください。

```
export function response(ctx) {
  return ctx.result
}
```

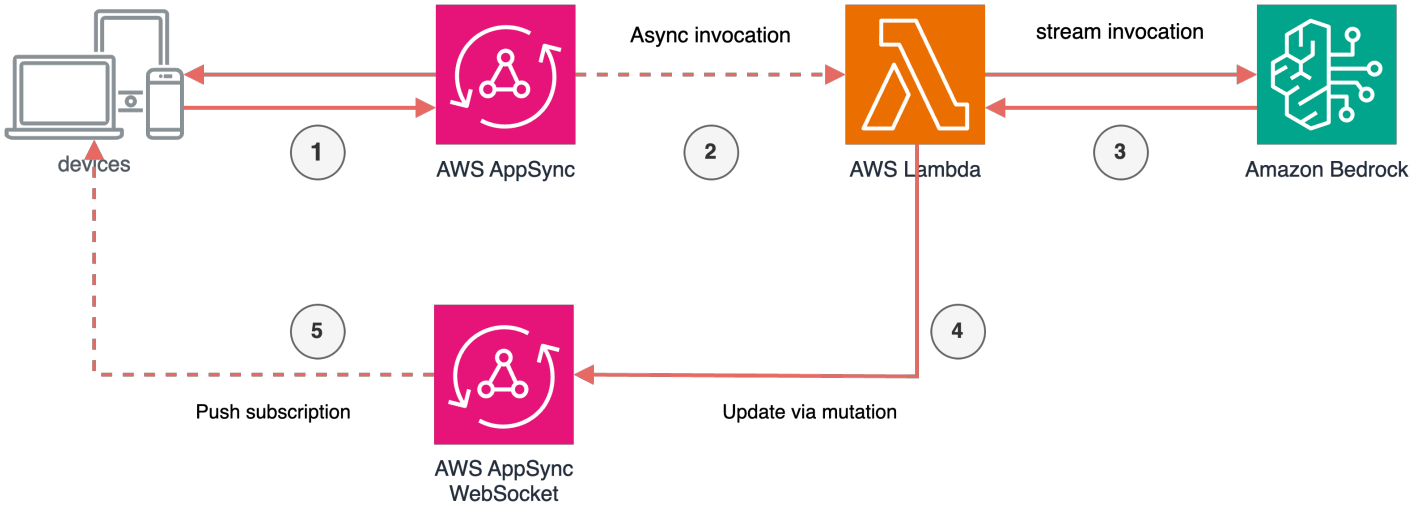
レスポンスオブジェクトに適用される形状の制限や必須フィールドはありません。ただし、GraphQL が厳密に型指定されているので、解決されたレスポンスは予想される GraphQL タイプに一致する必要があります。

長時間実行される呼び出し

現在、多くの組織は AI ゲートウェイ AWS AppSync として を使用して、Amazon Bedrock の基盤モデルを活用した生成 AI アプリケーションを構築しています。お客様は、WebSockets を利用したサブスクリプションを使用して AWS AppSync 、長時間実行されるモデル呼び出しからプログレッシブな更新を返します。これにより、非同期パターンを実装できます。

次の図は、このパターンを実装する方法を示しています。この図では、次のステップが実行されます。

1. クライアントは WebSocket をセットアップするサブスクリプションを開始し、生成 AI 呼び出しをトリガー AWS AppSync するように にリクエストします。
2. AWS AppSync はイベントモードで AWS Lambda 関数を呼び出し、すぐにクライアントにレスポンスを返します。
3. Lambda 関数は Amazon Bedrock でモデルを呼び出します。Lambda 関数は、 などの同期 API`InvokeModel`、または などのストリーム API を使用してInvokeModelWithResponseStream、プログレッシブ更新を取得できます。
4. 更新を受信するか、呼び出しが完了すると、Lambda 関数はミューテーションを介して更新を AWS AppSync API に送信し、サブスクリプションをトリガーします。
5. サブスクリプションイベントはリアルタイムで送信され、WebSocket 経由でクライアントによって受信されます。



タイプリファレンス

```
export type BedrockMessage = {
  role: 'user' | 'assistant' | string;
  content: BedrockMessageContent[];
};

export type BedrockMessageContent =
  | { text: string }
  | { guardContent: BedrockGuardContent }
  | { toolResult: BedrockToolResult }
  | { toolUse: BedrockToolUse };

export type BedrockGuardContent = {
  text: BedrockGuardContentText;
};

export type BedrockGuardContentText = {
  text: string;
  qualifiers?: ('grounding_source' | 'query' | 'guard_content' | string)[];
};

export type BedrockToolResult = {
  content: BedrockToolResultContent[];
  toolUseId: string;
  status?: string;
};
```

```
export type BedrockToolResultContent = { json: any } | { text: string };

export type BedrockToolUse = {
  input: any;
  name: string;
  toolUseId: string;
};

export type ConversePayload = {
  modelId: string;
  body: any;
  guardrailIdentifier?: string;
  guardrailVersion?: string;
  guardrailTrace?: string;
};

export type BedrockGuardrailConfig = {
  guardrailIdentifier: string;
  guardrailVersion: string;
  trace: string;
};

export type BedrockInferenceConfig = {
  maxTokens?: number;
  temperature?: number;
  stopSequences?: string[];
  topP?: number;
};

export type BedrockPromptVariableValues = {
  text: string;
};

export type BedrockToolConfig = {
  tools: BedrockTool[];
  toolChoice?: BedrockToolChoice;
};

export type BedrockTool = {
  toolSpec: BedrockToolSpec;
};

export type BedrockToolSpec = {
  name: string;
```

```
  description?: string;
  inputSchema: BedrockInputSchema;
};

export type BedrockInputSchema = {
  json: any;
};

export type BedrockToolChoice =
  | { tool: BedrockSpecificToolChoice }
  | { auto: any }
  | { any: any };

export type BedrockSpecificToolChoice = {
  name: string;
};

export type BedrockSystemContent =
  | { guardContent: BedrockGuardContent }
  | { text: string };

export type BedrockConverseOutput = {
  message?: BedrockMessage;
};

export type BedrockConverseMetrics = {
  latencyMs: number;
};

export type BedrockTokenUsage = {
  inputTokens: number;
  outputTokens: number;
  totalTokens: number;
};

export type BedrockConverseTrace = {
  guardrail?: BedrockGuardrailTraceAsssessment;
};

export type BedrockGuardrailTraceAsssessment = {
  inputAssessment?: { [key: string]: BedrockGuardrailAssessment };
  modelOutput?: string[];
  outputAssessments?: { [key: string]: BedrockGuardrailAssessment };
};
```

```
export type BedrockGuardrailAssessment = {
  contentPolicy?: BedrockGuardrailContentPolicyAssessment;
  contextualGroundingPolicy?: BedrockGuardrailContextualGroundingPolicyAssessment;
  invocationMetrics?: BedrockGuardrailInvocationMetrics;
  sensitiveInformationPolicy?: BedrockGuardrailSensitiveInformationPolicyAssessment;
  topicPolicy?: BedrockGuardrailTopicPolicyAssessment;
  wordPolicy?: BedrockGuardrailWordPolicyAssessment;
};

export type BedrockGuardrailContentPolicyAssessment = {
  filters: BedrockGuardrailContentFilter[];
};

export type BedrockGuardrailContentFilter = {
  action: 'BLOCKED' | string;
  confidence: 'NONE' | 'LOW' | 'MEDIUM' | 'HIGH' | string;
  type:
    | 'INSULTS'
    | 'HATE'
    | 'SEXUAL'
    | 'VIOLENCE'
    | 'MISCONDUCT'
    | 'PROMPT_ATTACK'
    | string;
  filterStrength: 'NONE' | 'LOW' | 'MEDIUM' | 'HIGH' | string;
};

export type BedrockGuardrailContextualGroundingPolicyAssessment = {
  filters: BedrockGuardrailContextualGroundingFilter;
};

export type BedrockGuardrailContextualGroundingFilter = {
  action: 'BLOCKED' | 'NONE' | string;
  score: number;
  threshold: number;
  type: 'GROUNDING' | 'RELEVANCE' | string;
};

export type BedrockGuardrailInvocationMetrics = {
  guardrailCoverage?: BedrockGuardrailCoverage;
  guardrailProcessingLatency?: number;
  usage?: BedrockGuardrailUsage;
};
```

```
export type BedrockGuardrailCoverage = {
  textCharacters?: BedrockGuardrailTextCharactersCoverage;
};

export type BedrockGuardrailTextCharactersCoverage = {
  guarded?: number;
  total?: number;
};

export type BedrockGuardrailUsage = {
  contentPolicyUnits: number;
  contextualGroundingPolicyUnits: number;
  sensitiveInformationPolicyFreeUnits: number;
  sensitiveInformationPolicyUnits: number;
  topicPolicyUnits: number;
  wordPolicyUnits: number;
};

export type BedrockGuardrailSensitiveInformationPolicyAssessment = {
  piiEntities: BedrockGuardrailPiiEntityFilter[];
  regexes: BedrockGuardrailRegexFilter[];
};

export type BedrockGuardrailPiiEntityFilter = {
  action: 'BLOCKED' | 'ANONYMIZED' | string;
  match: string;
  type:
    | 'ADDRESS'
    | 'AGE'
    | 'AWS_ACCESS_KEY'
    | 'AWS_SECRET_KEY'
    | 'CA_HEALTH_NUMBER'
    | 'CA_SOCIAL_INSURANCE_NUMBER'
    | 'CREDIT_DEBIT_CARD_CVV'
    | 'CREDIT_DEBIT_CARD_EXPIRY'
    | 'CREDIT_DEBIT_CARD_NUMBER'
    | 'DRIVER_ID'
    | 'EMAIL'
    | 'INTERNATIONAL_BANK_ACCOUNT_NUMBER'
    | 'IP_ADDRESS'
    | 'LICENSE_PLATE'
    | 'MAC_ADDRESS'
    | 'NAME'
```

```
    | 'PASSWORD'
    | 'PHONE'
    | 'PIN'
    | 'SWIFT_CODE'
    | 'UK_NATIONAL_HEALTH_SERVICE_NUMBER'
    | 'UK_NATIONAL_INSURANCE_NUMBER'
    | 'UK_UNIQUE_TAXPAYER_REFERENCE_NUMBER'
    | 'URL'
    | 'USERNAME'
    | 'US_BANK_ACCOUNT_NUMBER'
    | 'US_BANK_ROUTING_NUMBER'
    | 'US_INDIVIDUAL_TAX_IDENTIFICATION_NUMBER'
    | 'US_PASSPORT_NUMBER'
    | 'US_SOCIAL_SECURITY_NUMBER'
    | 'VEHICLE_IDENTIFICATION_NUMBER'
    | string;
};

export type BedrockGuardrailRegexFilter = {
  action: 'BLOCKED' | 'ANONYMIZED' | string;
  match?: string;
  name?: string;
  regex?: string;
};

export type BedrockGuardrailTopicPolicyAssessment = {
  topics: BedrockGuardrailTopic[];
};

export type BedrockGuardrailTopic = {
  action: 'BLOCKED' | string;
  name: string;
  type: 'DENY' | string;
};

export type BedrockGuardrailWordPolicyAssessment = {
  customWords: BedrockGuardrailCustomWord[];
  managedWordLists: BedrockGuardrailManagedWord[];
};

export type BedrockGuardrailCustomWord = {
  action: 'BLOCKED' | string;
  match: string;
};
```

```
export type BedrockGuardrailManagedWord = {
  action: 'BLOCKED' | string;
  match: string;
  type: 'PROFANITY' | string;
};
```

AWS AppSync リゾルバーマッピングテンプレートリファレンス (VTL)

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

以下のセクションでは、マッピングテンプレートでユーティリティ操作を使用する方法について説明します。

- リゾルバーマッピングテンプレートの概要 - リゾルバーの AWS AppSyncでの仕組みの詳細について説明します。
- リゾルバーマッピングテンプレートプログラミングガイド - 基本的な VTL データ構造とロジック処理について説明します。
- リゾルバーマッピングテンプレートコンテキストリファレンス - コンテキストマップの詳細とリゾルバーでの使用方法について説明します。
- リゾルバーマッピングテンプレートユーティリティリファレンス - ユーティリティを使用してコードを簡素化する方法について説明します。
- DynamoDB 用リゾルバーマッピングテンプレートリファレンス - リゾルバーが DynamoDB とやり取りする方法について詳しく説明します。
- RDS 用リゾルバーマッピングテンプレートリファレンス - リゾルバー構造と RDS とのやり取りの詳細について説明します。
- OpenSearch 用リゾルバーマッピングテンプレートリファレンス - リゾルバーのリクエストとレスポンスの構造、および OpenSearch Service とのやり取りの詳細について説明します。
- Lambda 用リゾルバーマッピングテンプレートリファレンス - リゾルバーのリクエストとレスポンスの構造、および Lambda とのやり取りの詳細について説明します。
- EventBridge 用リゾルバーマッピングテンプレートリファレンス - リゾルバーのリクエストとレスポンスの構造、および EventBridge とのやり取りの詳細について説明します。
- None データソース用リゾルバーマッピングテンプレートリファレンス - リゾルバーのリクエストとレスポンスの構造、および NONE データソースとのやり取りの詳細について説明します。
- HTTP 用リゾルバーマッピングテンプレートリファレンス - リゾルバーのリクエストとレスポンスの構造、および HTTP エンドポイントとのやり取りの詳細について説明します。

AWS AppSync リゾルバーマッピングテンプレートの概要

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync では、 リソースに対してオペレーションを実行して GraphQL リクエストに応答できます。クエリ や ミューテーション など、実行する GraphQL フィールドごとに、データソースとやり取りするためにリゾルバーをアタッチする必要があります。通常、この通信はパラメータを介して行うか、データソースに固有の処理を使用します。

リゾルバーは GraphQL とデータソースの間のコネクタです。受信した GraphQL リクエストをバックエンドデータソースの手順に変換する方法と、そのデータソースからのレスポンスを GraphQL レスポンスに変換する方法を AWS AppSync に伝えます。これらは Apache Velocity Template Language (VTL) で記述されます。これにより、入力としてリクエストを受け取り、リゾルバーに関する指示を含む JSON ドキュメントを出力します。GraphQL フィールドから引数を渡すなどの簡単な指示や、引数の指定により、項目をビルドしてから DynamoDB への挿入を繰り返すといったより複雑な指示をマッピングテンプレートを使用して行えます。

AWS AppSync には、マッピングテンプレートをわずかに異なる方法で活用する 2 種類のリゾルバーがあります。

- ユニットリゾルバー
- パイプラインリゾルバー

ユニットリゾルバー

ユニットリゾルバーは、リクエストおよびレスポンステンプレートのみを含む自己完結型エンティティです。1 つのデータソースから項目を一覧表示するなどのシンプルな 1 つのオペレーションに、この種類のリゾルバーを使用します。

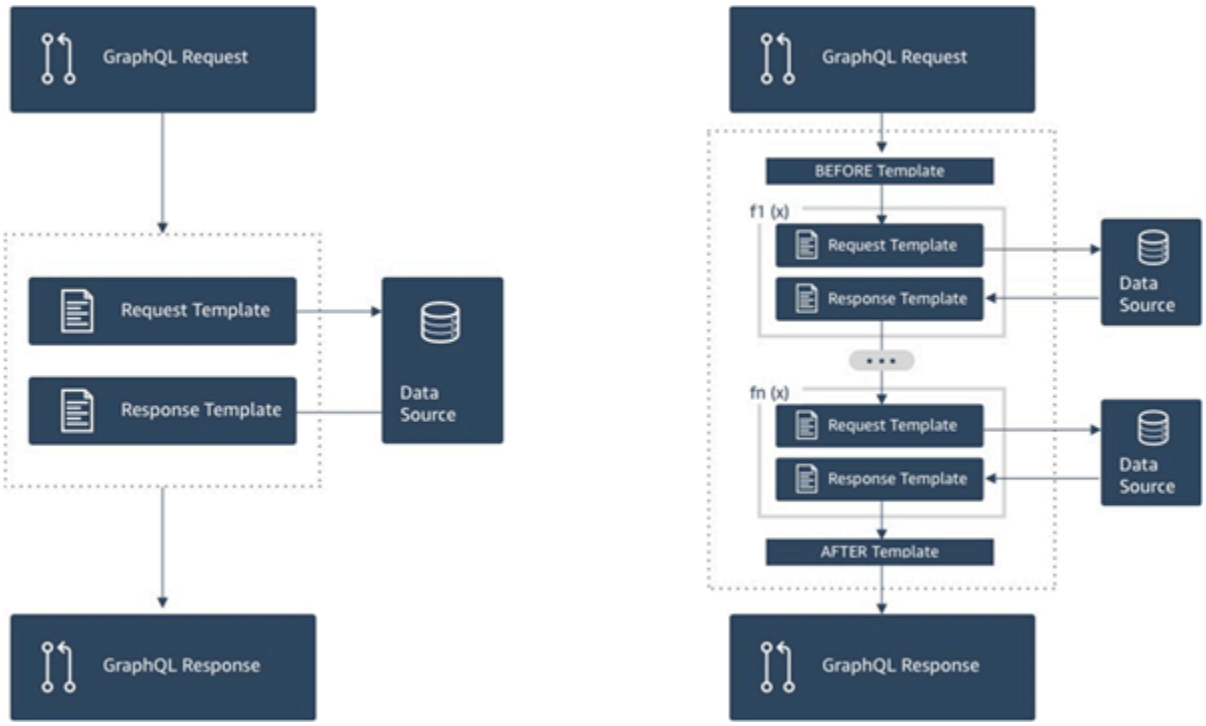
- リクエストテンプレート: GraphQL オペレーションが解析された後に受信リクエストを受け取り、選択されたデータソースオペレーション用のリクエスト設定に変換します。
- レスポンステンプレート: データソースからのレスポンスを解釈し、GraphQL フィールド出力タイプの形式にマッピングします。

パイプラインリゾルバー

パイプラインリゾルバーには、順番に実行される 1 つ以上の関数が含まれています。各関数には、リクエストテンプレートとレスポンステンプレートが含まれています。パイプラインのリゾルバーには、一連の関数を囲むテンプレートである before テンプレートとafter テンプレートも含まれています。After テンプレートは GraphQL フィールド出力タイプにマッピングします。パイプラインのリゾルバーは、レスポンステンプレートの出力をマッピングする点でユニットリゾルバーとは異なります。パイプラインリゾルバーは、別の関数の入力や、パイプラインリゾルバーのafterテンプレートを含むどんな出力でもマッピングできます。

パイプラインリゾルバー機能では、スキーマ内の複数のリゾルバーにまたがって再利用できる共通のロジックを作成できます。関数はデータソースに直接アタッチされており、ユニットリゾルバーのものと同じリクエストおよびレスポンスマッピングテンプレート形式を含んでいます。

次の図は、左側がユニットリゾルバー、右側がパイプラインリゾルバーのプロセスフローを示しています。



パイプラインリゾルバーには、少々複雑になるものの、ユニットリゾルバーが提供する機能の上位集合が含まれています。

パイプラインリゾルバーの仕組み

パイプラインリゾルバーは、Before マッピングテンプレート、After マッピングテンプレート、関数のリストで構成されています。各関数には、データソースに対して実行されるリクエストおよびレスポンスマッピングテンプレートがあります。パイプラインリゾルバーは実行をリスト内の関数に委任するため、いずれのデータソースにもリンクされていません。ユニットリゾルバーと関数は、データソースに対してオペレーションを実行するプリミティブです。詳細については、「リゾルバーのマッピングテンプレートの概要」を参照してください。

Before マッピングテンプレート

パイプラインリゾルバーのリクエストマッピングテンプレート (Before ステップとも呼ばれる) では、定義した関数を実行する前に準備ロジックを実行できます。

関数のリスト

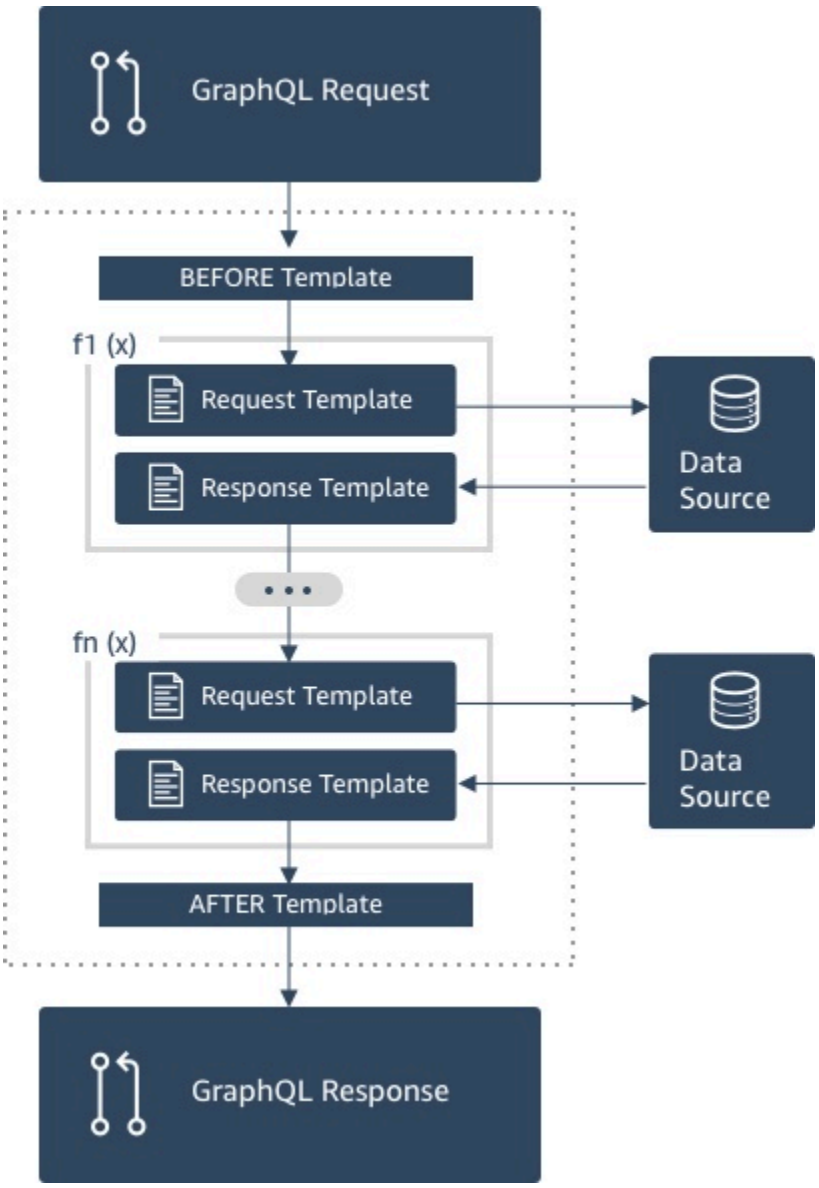
パイプラインリゾルバーが順番に実行する関数のリスト。パイプラインリゾルバーのリクエストマッピングテンプレートの評価結果は、最初の関数で `$ctx.prev.result` として使用可能になります。各関数の出力は、以下の関数で `$ctx.prev.result` として使用可能です。

After マッピングテンプレート

パイプラインリゾルバーのレスポンスマッピングテンプレート (After ステップとも呼ばれる) では、最後の関数の出力から、想定される GraphQL フィールドタイプへの、最終的なマッピングロジックを実行できます。関数のリストの最後にある関数の出力は、パイプラインリゾルバーのマッピングテンプレートで `$ctx.prev.result` または `$ctx.result` として使用可能です。

実行フロー

2 つの関数で構成されるパイプラインリゾルバーがあるとします。以下のリストでは、リゾルバーが呼び出されたときの実行フローを表しています。



1. パイプラインリゾルバーの Before マッピングテンプレート
2. 関数 1: 関数のリクエストマッピングテンプレート
3. 関数 1: データソースの呼び出し
4. 関数 1: 関数のレスポンスマッピングテンプレート
5. 関数 2: 関数のリクエストマッピングテンプレート
6. 関数 2: データソースの呼び出し
7. 関数 2: 関数のレスポンスマッピングテンプレート
8. パイプラインリゾルバーの After マッピングテンプレート

> **Note**
>
> パイプラインリゾルバーの実行フローは単方向であり、リゾルバーで静的に定義されています。

便利な Apache Velocity Template Language (VTL) ユーティリティ

アプリケーションの複雑さが増すにつれて、開発生産性を促進するのが、VTL ユーティリティとディレクティブの目的です。パイプラインリゾルバーでの作業には、以下のユーティリティが役立ちます。

$ctx.stash

stash は、リゾルバーと関数の各マッピングテンプレート内で使用可能になる `Map` です。同じ stash インスタンスは 1 つのリゾルバーの実行を通して存続します。つまり、stash を使用して、リクエストとレスポンスのマッピングテンプレート間、およびパイプラインリゾルバーの関数間で、任意のデータを渡すことができます。stash は Java Map データ構造と同じメソッドを継承しています。

$ctx.prev.result

`$ctx.prev.result` は、パイプラインリゾルバーで実行された前のオペレーションの結果を表します。

前のオペレーションがパイプラインリゾルバーの Before マッピングテンプレートであった場合、`$ctx.prev.result` はテンプレートの評価の出力を表し、パイプライン内の最初の関数に使用可能になります。前のオペレーションが最初の関数であった場合、`$ctx.prev.result` は最初の関数の出力を表し、パイプライン内の 2 番目の関数に使用可能になります。前のオペレーションが最後の関数であった場合、`$ctx.prev.result` は最後の関数の出力を表し、パイプラインリゾルバーの After マッピングテンプレートに使用可能になります。

#return(data: Object)

マッピングテンプレートから途中で戻る必要がある場合は、`#return(data: Object)` ディレクティブが便利です。`#return(data: Object)` は、プログラミング言語の return キーワードと似ています。これは、最も近いスコープのロジックブロックから戻るためです。つまり、リゾルバーのマッピングテンプレート内で `#return` を使用すると、リゾルバーから戻ることになります。リゾルバーのマッピングテンプレートで `#return(data: Object)` を使用すると、GraphQL フィールドに `data` が設定されます。さらに、関数のマッピングテンプレートから `#return(data: Object)`

を使用すると、実行が関数から戻り、パイプライン内の次の関数に、またはリゾルバーのレスポンスマッピングテンプレートに継続されます。

#return

これは `#return(data: Object)` と同じですが、代わりに `null` が返されます。

$util.error

`$util.error` ユーティリティはフィールドエラーをスローするのに便利です。関数のマッピングテンプレート内で `$util.error` を使用すると、フィールドエラーがすぐにスローされ、それ以降の関数が実行されなくなります。詳細およびその他の `$util.error` 署名については、「リゾルバーのマッピングテンプレートのユーティリティリファレンス」を参照してください。

$util.appendError

注意: `$util.appendError` は `$util.error()` と似ていますが、マッピングテンプレートの評価を中断しないという違いがあります。代わりに、フィールドにエラーがあったことを通知します。ただし、テンプレートを評価し、データを引き続き返すことも可能です。関数内で `$util.appendError` を使用しても、パイプラインの実行フローは中断されません。詳細およびその他の `$util.error` 署名については、「リゾルバーのマッピングテンプレートのユーティリティリファレンス」を参照してください。

サンプルテンプレート

DynamoDB データソースがあり、また次の GraphQL クエリにより `Post` 型を返す `getPost(id:ID!)` というフィールドの ユニット リゾルバーがあるとします。

```
getPost(id:1){
    id
    title
    content
}
```

リゾルバーのテンプレートは以下のようになります。

```
{
    "version" : "2018-05-29",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
```

```
    }
}
```

これは、id の入力パラメータ値 1 を ${ctx.args.id} に置き換え、次の JSON を生成します。

```
{
    "version" : "2018-05-29",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "1" }
    }
}
```

AWS AppSync はこのテンプレートを使用して、DynamoDB と通信し、データを取得する (または必要に応じて他のオペレーションを実行する) 手順を生成します。データが返されたら、 AWS AppSync はオプションのレスポンスマッピングテンプレートを使用してこの指示を実行します。このテンプレートは、データの形式操作または口ジックを実行するために使用できます。たとえば、DynamoDB から次のような結果を取得します。

```
{
        "id" : 1,
        "theTitle" : "AWS AppSync works offline!",
        "theContent-part1" : "It also has realtime functionality",
        "theContent-part2" : "using GraphQL"
}
```

次のようなレスポンスマッピングテンプレートを使用して、2 つのフィールドを 1 つのフィールドに結合することもできます。

```
{
        "id" : $util.toJson($context.data.id),
        "title" : $util.toJson($context.data.theTitle),
        "content" : $util.toJson("${context.data.theContent-part1}
 ${context.data.theContent-part2}")
}
```

テンプレートがデータに適用されると、データは次のように成形されます。

```
{
        "id" : 1,
```

```
        "title" : "AWS AppSync works offline!",
        "content" : "It also has realtime functionality using GraphQL"
}
```

このデータは以下のように、クライアントへのレスポンスとして返されます。

```
{
        "data": {
                "getPost":      {
                        "id" : 1,
                        "title" : "AWS AppSync works offline!",
                        "content" : "It also has realtime functionality using GraphQL"
                }
        }
}
```

ほとんどの場合、レスポンスマッピングテンプレートはデータを単純に転送します。個々の項目や項目のリストを返す場合は、大きく異なります。個々の項目について、パススルーは次のようになります。

```
$util.toJson($context.result)
```

リストについて、パススルーは通常次のようになります。

```
$util.toJson($context.result.items)
```

ユニットリゾルバーとパイプラインリゾルバーの両方のさらに多くの例については、リゾルバーのチュートリアルを参照してください。

評価されたマッピングテンプレートの逆シリアル化ルール

マッピングテンプレートは文字列として評価されます。In AWS AppSync では、出力文字列が有効になるには JSON 構造に従う必要があります。

さらに、以下の逆シリアル化ルールが適用されます。

JSON オブジェクトでは、重複キーは使用できません。

評価されたマッピングテンプレート文字列が JSON オブジェクトを表すか、重複するキーのあるオブジェクトを含む場合、マッピングテンプレートから以下のエラーメッセージが返されます。

```
Duplicate field 'aField' detected on Object. Duplicate JSON keys are not allowed.
```

評価されたリクエストマッピングテンプレートの重複キーの例:

```
{
    "version": "2018-05-29",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "1",
        "field": "getPost" ## key 'field' has been redefined
    }
}
```

このエラーを修正するには、JSON オブジェクトのキーを再定義しないでください。

JSON オブジェクトでは末尾に文字は使用できません。

評価されたマッピングテンプレート文字列が JSON オブジェクトを表し、末尾に無関係な文字が含まれている場合は、マッピングテンプレートから以下のエラーメッセージが返されます。

```
Trailing characters at the end of the JSON string are not allowed.
```

評価されたリクエストマッピングテンプレートの末尾の文字の例:

```
{
    "version": "2018-05-29",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "1",
    }
}extraneouschars
```

このエラーを修正するには、評価対象のテンプレートが厳密に JSON に評価されるようにしてください。

AWS AppSync リゾルバーマッピングテンプレートプログラミングガイド

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

これは、Apache Velocity Template Language (VTL) を使用したプログラミングのクックブック形式のチュートリアルです AWS AppSync。JavaScript、C、Java など他のプログラミング言語に慣れている場合は、かなり容易に進められます。

AWS AppSync は VTL を使用して、クライアントからの GraphQL リクエストをデータソースへのリクエストに変換します。その後、このプロセスを逆転して、データソースレスポンスを GraphQL レスポンスに変換します。VTL はロジカルなテンプレート言語で、ウェブアプリケーションの標準的なリクエスト/レスポンスフローで、次のような手法を利用した、リクエストとレスポンスの両方を操作する機能が用意されています。

- 新しい項目のデフォルト値
- 入力の検証とフォーマット
- データ変換および整形
- リスト、マップ、配列の値を取り出す、または変更するための繰り返し処理
- ユーザー ID に基づくレスポンスのフィルタリング/変更
- 複合認可チェック

たとえば、GraphQL 引数でサービスの電話番号の検証を実行する、または DynamoDB に格納する前に入力パラメータを大文字に変換する場合を考えます。または、クライアントシステムで、GraphQL 引数、JWT トークンクレーム、または HTTP ヘッダーの一部としてコードを渡し、コードがリストの特定の文字列と一致する場合にのみ、データを返す場合を考えます。これらはすべて、 AWS AppSync の VTL で実行できる論理チェックです。

VTL では、使い慣れているプログラミング手法を使用してロジックを適用できます。ただし、標準的なリクエスト/レスポンスフロー内での実行に限られています。これにより、ユーザーベースが拡大しても、GraphQL API のスケーラビリティが確保されます。 AWS AppSync はリゾルバーとし

て もサポート AWS Lambda しているため、柔軟性が必要な場合は、選択したプログラミング言語 (Node.js、Python、Go、Java など) で Lambda 関数を記述できます。

セットアップ

プログラミング言語を学ぶ一般的な手法は、結果 (たとえば、JavaScript の `console.log(variable)`) を出力し、何が起こるのかを確認することです。このチュートリアルでは、シンプルな GraphQL スキーマを作成し、Lambda 関数に値のマップを渡すことで、このデモを実行します。Lambda 関数で値を出力し、それらに応答します。これにより、リクエスト/レスポンスフローを理解し、さまざまなプログラミング手法を確認できます。

以下の GraphQL スキーマを作成することから開始します。

```
type Query {
    get(id: ID, meta: String): Thing
}

type Thing {
    id: ID!
    title: String!
    meta: String
}

schema {
    query: Query
}
```

ここで、Node.js を言語として使用して、次の AWS Lambda 関数を作成します。

```
exports.handler = (event, context, callback) => {
    console.log('VTL details: ', event);
    callback(null, event);
};
```

AWS AppSync コンソールのデータソースペインで、この Lambda 関数を新しいデータソースとして追加します。コンソールの [スキーマ] ページに戻り、右側で、`get(...):Thing` クエリの横にある [アタッチ] ボタンをクリックします。リクエストテンプレートでは、[Invoke and forward arguments (引数の呼び出しと転送)] メニューから既存のテンプレートを選択します。レスポンステンプレートでは、[Return Lambda result (Lambda 関数の結果を返す)] を選択します。

Lambda 関数の Amazon CloudWatch Logs を 1 か所で開き、 AWS AppSync コンソールのクエリタブから次の GraphQL クエリを実行します。

```
query test {
  get(id:123 meta:"testing"){
    id
    meta
  }
}
```

GraphQL のレスポンスには、id:123 と meta:testing が含まれます。Lambda 関数がエコーバックするためです。数秒後、CloudWatch Logs でこれらの詳細とともにレコードが表示されます。

[変数]

VTL では参照を使用します。これらの参照を使用してデータを保存または操作できます。VTL には 3 種類の参照型 (変数、プロパティ、およびメソッド) があります。変数には、先頭に $ 記号が付き、#set ディレクティブで作成されます。

```
#set($var = "a string")
```

変数は、他の言語で使い慣れた、同様の型 (数値、文字列、配列、リスト、マップなど) に保存します。Lambda リゾルバーのデフォルトリクエストテンプレートで JSON ペイロードが送信されていることがわかります。

```
"payload": $util.toJson($context.arguments)
```

ここで注意すべき点がいくつかあります。まず、 AWS AppSync は一般的なオペレーションのためのいくつかの便利な関数を提供します。この例では、$util.toJson は変数を JSON に変換します。次に、変数 $context.arguments は GraphQL リクエストからマップオブジェクトとして自動的に入力されます。新しいマップを次のように作成できます。

```
#set( $myMap = {
  "id": $context.arguments.id,
  "meta": "stuff",
  "upperMeta" : $context.arguments.meta.toUpperCase()
} )
```

$myMap という変数が作成されました。これには id、meta、および upperMeta のキーがあります。これは、以下の特徴も示しています。

- id には、GraphQL 引数からキーが入力されます。これは VTL で一般的で、クライアントから引数を取得するためのものです。
- meta は、値がハードコードされ、デフォルト値を示します。
- upperMeta は meta 引数を .toUpperCase() メソッドを使用して変換します。

リクエストテンプレートの先頭に以前のコードを配置して、payload を変更して、新しい $myMap 変数を使用するようにします。

```
"payload": $util.toJson($myMap)
```

Lambda 関数を実行し、CloudWatch のログのデータとともに、レスポンスの変更を確認できます。このチュートリアルの残りの部分を実行するので、同様のテストを実行できるように $myMap の入力を保持します。

変数に properties_ を設定することもできます。これらは単純な文字列、配列、または JSON です。

```
#set($myMap.myProperty = "ABC")
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])
#set($myMap.jsonProperty = {
    "AppSync" : "Offline and Realtime",
    "Cognito" : "AuthN and AuthZ"
})
```

静的参照

VTL がテンプレート作成言語であるため、デフォルトでは、そこで使用するすべての参照は .toString() を実行します。参照が未定義の場合、実際の参照表現を、文字列として出力します。例:

```
#set($myValue = 5)
##Prints '5'
$myValue

##Prints '$somethingelse'
$somethingelse
```

これに対応するために、VTL には「静的参照」、または「サイレント参照」構文があり、この動作を抑制するかどうかをテンプレートエンジンに指示します。この構文は $!{} です。たとえば、前のコードで $!{somethingelse} を使用してわずかに変更した場合、印刷は抑制されます。

```
#set($myValue = 5)
##Prints '5'
$myValue

##Nothing prints out
$!{somethingelse}
```

メソッドの呼び出し

前の例では、変数を作成し、同時に値を設定する方法を説明しました。次に示すように、データをマップに追加して、これを 2 つのステップで実行することもできます。

```
#set ($myMap = {})
#set ($myList = [])

##Nothing prints out
$!{myMap.put("id", "first value")}
##Prints "first value"
$!{myMap.put("id", "another value")}
##Prints true
$!{myList.add("something")}
```

ただし、この動作について留意すべき点があります。静的参照表記 `$!{}` で、上記のようにメソッドを呼び出すことができますが、実行されたメソッドの戻り値は抑制されません。そのため `##Prints "first value"` と `##Prints true` が追加されています。キーがすでに存在している場合に、値を挿入するなど、リストやマップで反復処理をしているときに、エラーが発生する場合があります。出力で評価時に予期しない文字列がテンプレートに追加されるためです。

これに対する回避策は、`#set` ディレクティブを使用してメソッドを呼び出し、変数を無視するとうまくいくことがあります。以下に例を示します。

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

この手法をテンプレートで使用すると、予期しない文字列がテンプレートに出力されなくなります。AWS AppSync は、より簡潔な表記で同じ動作を提供する代替の便利な関数を提供します。これにより、これらの実装の詳細を検討する必要がなくなります。`$util.quiet()` またはその エイリアス `$util.qr()` でこの関数にアクセスできます。以下に例を示します。

```
#set ($myMap = {})
```

```
#set ($myList = [])

##Nothing prints out
$util.quiet($myMap.put("id", "first value"))
##Nothing prints out
$util.qr($myList.add("something"))
```

文字列

多くのプログラミング言語の場合と同じように、文字列は、特に変数から構築する場合に対処が困難なことがあります。VTL で発生する共通の事項がいくつかあります。

DynamoDB のようなデータソースに文字列としてデータを挿入する場合、GraphQL 引数などの変数から入力されます。文字列は、二重引用符で囲まれ、文字列で変数を参照するために必要なのは "${}" だけです (静的参照表記の ！ がない場合)。これは JavaScript のテンプレートリテラル (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals) に似ています。

```
#set($firstname = "Jeff")
$!{myMap.put("Firstname", "${firstname}")}
```

GraphQL クライアントから引数を使用するときの "author": { "S" : "${context.arguments.author}"}、または "id" : { "S" : "$util.autoId()"} のような ID 自動生成などの DynamoDB リクエストテンプレートでこれを確認できます。つまり、データを入力するため、文字列の中でメソッドの結果または変数を参照できるということです。

部分文字列を取り出すなど、Java String クラスのパブリックメソッドを使用することもできます。

```
#set($bigstring = "This is a long string, I want to pull out everything after the
 comma")
#set ($comma = $bigstring.indexOf(','))
#set ($comma = $comma +2)
#set ($substring = $bigstring.substring($comma))

$util.qr($myMap.put("substring", "${substring}"))
```

文字列連結も非常に一般的なタスクです。これを行うには、変数参照を単独で、または静的な値とともに使用します。

```
#set($s1 = "Hello")
```

```
#set($s2 = " World")

$util.qr($myMap.put("concat","$s1$s2"))
$util.qr($myMap.put("concat2","Second $s1 World"))
```

loop

変数を作成して、メソッドを呼び出したので、ロジックをコードに追加できます。他の言語とは異なり、VTL では反復回数が事前に定義されている場合にのみ、ループが許可されます。Velocity には do..while はありません。この設計により、評価プロセスは常に確実に終了します。これは GraphQL オペレーションを実行するときに、スケーラビリティのための境界になります。

ループは #foreach で作成され、ループ変数や、配列、リスト、マップ、コレクションなど反復可能オブジェクトの入力が必要です。#foreach ループの典型的なプログラミング例では、コレクションの項目をループし、出力します。このケースでは、それらを取り出し、マップに追加します。

```
#set($start = 0)
#set($end = 5)
#set($range = [$start..$end])

#foreach($i in $range)
   ##$util.qr($myMap.put($i, "abc"))
   ##$util.qr($myMap.put($i, $i.toString()+"foo")) ##Concat variable with string
   $util.qr($myMap.put($i, "${i}foo"))     ##Reference a variable in a string with
 "${varname}"
#end
```

この例では、いくつかの注意点を示します。まず、範囲 [..] 演算子で変数を使用して、反復可能オブジェクトを作成します。次に、各項目は、操作できる $i 変数によって参照されます。前の例では、コメントは、二重の ## 記号で示されます。これは、文字列を使用した、異なるメソッドの連結であるとともに、キー (値) の両方に、ループ変数を使用する例でもあります。

$i は整数であることに注意してください。.toString() メソッドを呼び出すことができます。GraphQL の INT タイプでは、これが役に立ちます。

範囲演算子を直接使用することもできます。次に例を示します。

```
#foreach($item in [1..5])
    ...
#end
```

配列

ここまで、マップを操作しましたが、配列も VTL で一般的です。配列では、.isEmpty()、.size()、.set()、.get()、.add() などのいくつかの基本メソッドで、以下のようにアクセスできます。

```
#set($array = [])
#set($idx = 0)

##adding elements
$util.qr($array.add("element in array"))
$util.qr($myMap.put("array", $array[$idx]))

##initialize array vals on create
#set($arr2 = [42, "a string", 21, "test"])

$util.qr($myMap.put("arr2", $arr2[$idx]))
$util.qr($myMap.put("isEmpty", $array.isEmpty()))  ##isEmpty == false
$util.qr($myMap.put("size", $array.size()))

##Get and set items in an array
$util.qr($myMap.put("set", $array.set(0, 'changing array value')))
$util.qr($myMap.put("get", $array.get(0)))
```

前の例では配列インデックス表記を使用して arr2[$idx] の要素を取得しました。同様に、マップ/ディクショナリから名前で参照することができます。

```
#set($result = {
    "Author" : "Nadia",
    "Topic" : "GraphQL"
})

$util.qr($myMap.put("Author", $result["Author"]))
```

条件式を使用して、レスポンステンプレートでデータソースから返される結果をフィルタリングするときに、これがよく使われます。

条件チェック

前の #foreach のセクションで、VTL でデータを変換するロジックを使用するいくつかの例を示しました。データを評価する条件チェックがランタイムにも適用されます。

```
#if(!$array.isEmpty())
    $util.qr($myMap.put("ifCheck", "Array not empty"))
#else
    $util.qr($myMap.put("ifCheck", "Your array is empty"))
#end
```

上記のブール式の #if( ) チェックは、問題ありませんが、分岐に演算子と #elseif( ) を使用することもできます。

```
#if ($arr2.size() == 0)
    $util.qr($myMap.put("elseIfCheck", "You forgot to put anything into this array!"))
#elseif ($arr2.size() == 1)
    $util.qr($myMap.put("elseIfCheck", "Good start but please add more stuff"))
#else
    $util.qr($myMap.put("elseIfCheck", "Good job!"))
#end
```

これらの 2 つの例は否定 (!) と一致 (==) を示しています。||、&&、>、<、>=、<=、および != を使用することもできます。

```
#set($T = true)
#set($F = false)

#if ($T || $F)
  $util.qr($myMap.put("OR", "TRUE"))
#end

#if ($T && $F)
  $util.qr($myMap.put("AND", "TRUE"))
#end
```

注意 : Boolean.FALSE と null だけが条件で false と見なされます。ゼロ (0) と空の文字列 ("") は false に該当しません。

演算子

何らかの演算処理を実行する演算子がないと、プログラミング言語は完全とはいえません。手始めにいくつか例を紹介します。

```
#set($x = 5)
```

```
#set($y = 7)
#set($z = $x + $y)
#set($x-y = $x - $y)
#set($xy = $x * $y)
#set($xDIVy = $x / $y)
#set($xMODy = $x % $y)

$util.qr($myMap.put("z", $z))
$util.qr($myMap.put("x-y", $x-y))
$util.qr($myMap.put("x*y", $xy))
$util.qr($myMap.put("x/y", $xDIVy))
$util.qr($myMap.put("x|y", $xMODy))
```

ループと条件式を組み合わせて使用する

オブジェクトをループしアクションを実行する前にチェックを実行することは、データソースからの書き込みや読み込みの前など、VTL でデータを変換する際によく使われます。前のセクションのいくつかのツールを組み合わせることで、多くの機能が利用できます。1 つの便利な手法は #foreach では各項目に .count が自動的に使用できる点を理解することです。

```
#foreach ($item in $arr2)
  #set($idx = "item" + $foreach.count)
  $util.qr($myMap.put($idx, $item))
#end
```

たとえば、一定サイズ未満の値をマップから取り出す場合があります。#break ステートメントで、条件とともにカウントを使用してこれを実行できます。

```
#set($hashmap = {
  "DynamoDB" : "https://aws.amazon.com/dynamodb/",
  "Amplify" : "https://github.com/aws/aws-amplify",
  "DynamoDB2" : "https://aws.amazon.com/dynamodb/",
  "Amplify2" : "https://github.com/aws/aws-amplify"
})

#foreach ($key in $hashmap.keySet())
    #if($foreach.count > 2)
    #break
  #end
    $util.qr($myMap.put($key, $hashmap.get($key)))
#end
```

前の #foreach はマップに対して使用できる .keySet() で反復します。これにより、$key を取得し .get($key) で値を参照するためにアクセスできます。 AWS AppSync のクライアントからの GraphQL 引数はマップとして保存されます。また、.entrySet() で反復処理を実行でき、Set としてのキーと値の両方にアクセスでき、入力の検証や変換などの複雑な条件チェックを実行するか、他の変数を入力できます。

```
#foreach( $entry in $context.arguments.entrySet() )
#if ($entry.key == "XYZ" && $entry.value == "BAD")
    #set($myvar = "...")
  #else
    #break
  #end
#end
```

他の一般的な例は、データを同期するときの初期オブジェクトバージョンなど、デフォルト情報の自動入力です (競合の解決に非常に重要)。または認可チェック用のオブジェクトのデフォルト所有者もあります。Mary がこのブログ記事を作成した場合は以下のようになります。

```
#set($myMap.owner ="Mary")
#set($myMap.defaultOwners = ["Admins", "Editors"])
```

Context

AWS AppSync リゾルバーと VTL での論理チェック実行に慣れてきたところで、コンテキストオブジェクトを確認します。

```
$util.qr($myMap.put("context", $context))
```

これには、GraphQL リクエストの、アクセスできるすべての情報が含まれます。詳細な説明については、コンテキストリファレンスを参照してください。

フィルタリング

これまでチュートリアルで、Lambda 関数からのすべての情報が、非常にシンプルな JSON 変換と GraphQL クエリに返されます。

```
$util.toJson($context.result)
```

VTL ロジックは、データソースからレスポンスを取得するとき、特にリソースで許可チェックを実行するときに強力です。いくつかの例を見てみましょう。最初に、次のようにレスポンステンプレートの変更を試します。

```
#set($data = {
    "id" : "456",
    "meta" : "Valid Response"
})

$util.toJson($data)
```

GraphQL オペレーションで何が起こるかにかかわらず、ハードコードされた値がクライアントに返されます。meta フィールドに Lambda レスポンスから値が設定されるように少し変更し、条件式についての前のチュートリアルのように、elseIfCheck 値に設定します。

```
#set($data = {
    "id" : "456"
})

#foreach($item in $context.result.entrySet())
    #if($item.key == "elseIfCheck")
        $util.qr($data.put("meta", $item.value))
    #end
#end

$util.toJson($data)
```

$context.result はマップで、キーまたは値のいずれかで返されるロジックを実行するために entrySet() を使用できます。GraphQL オペレーションを実行したユーザーに関する情報が $context.identity に含まれているため、データソースから認可情報を返す場合は、ロジックに基づいて、ユーザーに返すデータ (すべて、一部、なし) を決定できます。次のようにレスポンステンプレートを変更します。

```
#if($context.result["id"] == 123)
    $util.toJson($context.result)
  #else
    $util.unauthorized()
#end
```

GraphQL クエリを実行した場合、データがノーマルとして返されます。ただし、id 引数を 123 以外のものに変更した場合 (`query test { get(id:456 meta:"badrequest"){} }`)、認可失敗のメッセージが表示されます。

認可ユースケースセクションに他の認可シナリオがあります。

テンプレートサンプル

このチュートリアルを実行していくと、ステップごとに拡張されて、このテンプレートになります。このテンプレートがまだない場合は、以下のテンプレートをテスト用にコピーします。

リクエストテンプレート

```
#set( $myMap = {
  "id": $context.arguments.id,
  "meta": "stuff",
  "upperMeta" : "$context.arguments.meta.toUpperCase()"
} )

##This is how you would do it in two steps with a "quiet reference" and you can use it
 for invoking methods, such as .put() to add items to a Map
#set ($myMap2 = {})
$util.qr($myMap2.put("id", "first value"))

## Properties are created with a dot notation
#set($myMap.myProperty = "ABC")
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])
#set($myMap.jsonProperty = {
    "AppSync" : "Offline and Realtime",
    "Cognito" : "AuthN and AuthZ"
})

##When you are inside a string and just have ${} without ! it means stuff inside curly
 braces are a reference
#set($firstname = "Jeff")
$util.qr($myMap.put("Firstname", "${firstname}"))

#set($bigstring = "This is a long string, I want to pull out everything after the
 comma")
#set ($comma = $bigstring.indexOf(','))
#set ($comma = $comma +2)
#set ($substring = $bigstring.substring($comma))
$util.qr($myMap.put("substring", "${substring}"))
```

```
##Classic for-each loop over N items:
#set($start = 0)
#set($end = 5)
#set($range = [$start..$end])
#foreach($i in $range)          ##Can also use range operator directly like
 #foreach($item in [1...5])
   ##$util.qr($myMap.put($i, "abc"))
   ##$util.qr($myMap.put($i, $i.toString()+"foo")) ##Concat variable with string
   $util.qr($myMap.put($i, "${i}foo"))     ##Reference a variable in a string with
 "${varname)"
#end

##Operators don't work
#set($x = 5)
#set($y = 7)
#set($z = $x + $y)
#set($x-y = $x - $y)
#set($xy = $x * $y)
#set($xDIVy = $x / $y)
#set($xMODy = $x % $y)
$util.qr($myMap.put("z", $z))
$util.qr($myMap.put("x-y", $x-y))
$util.qr($myMap.put("x*y", $xy))
$util.qr($myMap.put("x/y", $xDIVy))
$util.qr($myMap.put("x|y", $xMODy))

##arrays
#set($array = ["first"])
#set($idx = 0)
$util.qr($myMap.put("array", $array[$idx]))
##initialize array vals on create
#set($arr2 = [42, "a string", 21, "test"])
$util.qr($myMap.put("arr2", $arr2[$idx]))
$util.qr($myMap.put("isEmpty", $array.isEmpty()))  ##Returns false
$util.qr($myMap.put("size", $array.size()))
##Get and set items in an array
$util.qr($myMap.put("set", $array.set(0, 'changing array value')))
$util.qr($myMap.put("get", $array.get(0)))

##Lookup by name from a Map/dictionary in a similar way:
#set($result = {
    "Author" : "Nadia",
    "Topic" : "GraphQL"
```

```
})
$util.qr($myMap.put("Author", $result["Author"]))


##Conditional examples
#if(!$array.isEmpty())
$util.qr($myMap.put("ifCheck", "Array not empty"))
#else
$util.qr($myMap.put("ifCheck", "Your array is empty"))
#end

#if ($arr2.size() == 0)
$util.qr($myMap.put("elseIfCheck", "You forgot to put anything into this array!"))
#elseif ($arr2.size() == 1)
$util.qr($myMap.put("elseIfCheck", "Good start but please add more stuff"))
#else
$util.qr($myMap.put("elseIfCheck", "Good job!"))
#end

##Above showed negation(!) and equality (==), we can also use OR, AND, >, <, >=, <=,
 and !=
#set($T = true)
#set($F = false)
#if ($T || $F)
  $util.qr($myMap.put("OR", "TRUE"))
#end

#if ($T && $F)
  $util.qr($myMap.put("AND", "TRUE"))
#end

##Using the foreach loop counter - $foreach.count
#foreach ($item in $arr2)
  #set($idx = "item" + $foreach.count)
  $util.qr($myMap.put($idx, $item))
#end

##Using a Map and plucking out keys/vals
#set($hashmap = {
    "DynamoDB" : "https://aws.amazon.com/dynamodb/",
    "Amplify" : "https://github.com/aws/aws-amplify",
    "DynamoDB2" : "https://aws.amazon.com/dynamodb/",
    "Amplify2" : "https://github.com/aws/aws-amplify"
})
```

```
#foreach ($key in $hashmap.keySet())
    #if($foreach.count > 2)
        #break
    #end
    $util.qr($myMap.put($key, $hashmap.get($key)))
#end

##concatenate strings
#set($s1 = "Hello")
#set($s2 = " World")
$util.qr($myMap.put("concat","$s1$s2"))
$util.qr($myMap.put("concat2","Second $s1 World"))

$util.qr($myMap.put("context", $context))

{
    "version" : "2017-02-28",
    "operation": "Invoke",
    "payload": $util.toJson($myMap)
}
```

レスポンステンプレート

```
#set($data = {
"id" : "456"
})
#foreach($item in $context.result.entrySet())   ##$context.result is a MAP so we use
 entrySet()
    #if($item.key == "ifCheck")
        $util.qr($data.put("meta", "$item.value"))
    #end
#end

##Uncomment this out if you want to test and remove the below #if check
##$util.toJson($data)

#if($context.result["id"] == 123)
    $util.toJson($context.result)
  #else
    $util.unauthorized()
#end
```

AWS AppSync リゾルバーマッピングテンプレートコンテキストリファレンス

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync は、リゾルバーマッピングテンプレートを操作するための変数と関数のセットを定義します。これにより、GraphQL を使用してデータの論理的オペレーションが容易になります。このドキュメントでは、それらの関数について説明し、テンプレートの使用例を示します。

$context へのアクセス

$context 変数は、リゾルバー呼び出しのすべてのコンテキスト情報が保持されるマップです。この変数の構造は次のとおりです。

```
{
   "arguments" : { ... },
   "source" : { ... },
   "result" : { ... },
   "identity" : { ... },
   "request" : { ... },
   "info": { ... }
}
```

> **Note**
>
> キーによってディクショナリ/マップエントリ (context 内のエントリなど) にアクセスし、値を取得しようとしている場合は、Velocity Template Language (VTL) によって表記 <dictionary-element>.<key-name> を直接使用できます。ただし、キー名に特殊文字 (アンダースコア "_" など) が使われている場合など、一部のケースでは機能しない場合があります。常に <dictionary-element>.get("<key-name>") 表記を使用することをお勧めします。

$context マップの各フィールドは次のように定義されます。

$context フィールド

arguments

このフィールドのすべての GraphQL 引数を含むマップ。

identity

呼び出し元に関する情報を含むオブジェクト。このフィールドの構造の詳細については、「ID」を参照してください。

source

親フィールドの解像度を含むマップ。

stash

stash は、リゾルバーと関数の各マッピングテンプレート内で使用可能になるマップです。同じ stash インスタンスは 1 つのリゾルバーの実行を通して存続します。つまり、stash を使用して、リクエストとレスポンスのマッピングテンプレート間、およびパイプラインリゾルバーの関数間で、任意のデータを渡すことができます。stash は Java Map データ構造と同じメソッドを継承しています。

result

このリゾルバーの結果用のコンテナ。このフィールドはレスポンスマッピングテンプレートでのみ使用できます。

たとえば、次のクエリの `author` フィールドを解決する場合:

```
query {
    getPost(id: 1234) {
        postId
        title
        content
        author {
            id
            name
        }
    }
}
```

レスポンスマッピングテンプレートを処理する際に使用可能な完全な $context 変数は次のようになります。

```
{
  "arguments" : {
    id: "1234"
  },
  "source": {},
  "result" : {
      "postId": "1234",
      "title": "Some title",
      "content": "Some content",
      "author": {
        "id": "5678",
        "name": "Author Name"
      }
  },
  "identity" : {
      "sourceIp" : ["x.x.x.x"],
      "userArn" : "arn:aws:iam::123456789012:user/appsync",
      "accountId" : "666666666666",
      "user" : "AIDAAAAAAAAAAAAAAAAAA"
  }
}
```

prev.result

パイプラインリゾルバーで実行された前回のオペレーションの結果です。

前のオペレーションがパイプラインリゾルバーのリクエストマッピングテンプレートであった場合、$ctx.prev.result はテンプレートの評価の出力を表し、パイプライン内の最初の関数に使用可能になります。

前のオペレーションが最初の関数であった場合、$ctx.prev.result は最初の関数の出力を表し、パイプライン内の 2 番目の関数に使用可能になります。

前のオペレーションが最後の関数であった場合、$ctx.prev.result は最後の関数の出力を表し、パイプラインリゾルバーのレスポンスマッピングテンプレートに使用可能になります。

info

GraphQL リクエストに関する情報を含むオブジェクト。このフィールドの構造については、「情報」を参照してください。

アイデンティティ

identity セクションには、呼び出し元に関する情報が含まれています。このセクションのシェイプは、使用する AWS AppSync API の認可タイプによって異なります。

AWS AppSync セキュリティオプションの詳細については、「認可と認証」を参照してください。

API_KEY authorization

identity フィールドは入力されていません。

AWS_LAMBDA authorization

identityには、リクエストを承認する Lambda 関数によって返されるのと同じ resolverContext コンテンツを含む resolverContext キーが含まれています。

AWS_IAM authorization

identity の形式は次のとおりです。

```
{
    "accountId" : "string",
    "cognitoIdentityPoolId" : "string",
    "cognitoIdentityId" : "string",
    "sourceIp" : ["string"],
    "username" : "string", // IAM user principal
    "userArn" : "string",
    "cognitoIdentityAuthType" : "string", // authenticated/unauthenticated based on
 the identity type
    "cognitoIdentityAuthProvider" : "string" // the auth provider that was used to
 obtain the credentials
}
```

AMAZON_COGNITO_USER_POOLS authorization

identity の形式は次のとおりです。

```
{
    "sub" : "uuid",
    "issuer" : "string",
    "username" : "string"
    "claims" : { ... },
    "sourceIp" : ["x.x.x.x"],
    "defaultAuthStrategy" : "string"
```

```
}
```

各フィールドは次のように定義されています。

accountId

発信者の AWS アカウント ID。

claims

ユーザーが持っているクレーム。

cognitoIdentityAuthType

ID タイプに基づいて認証済みまたは未認証のいずれか。

cognitoIdentityAuthProvider

リクエストの署名に使用された認証情報の取得先となる外部 ID プロバイダ情報のカンマ区切りリスト。

cognitoIdentityId

リクエストを行う呼び出し元の Amazon Cognito 認証 ID。

cognitoIdentityPoolId

呼び出し元に関連付けられている Amazon Cognito ID プール。

defaultAuthStrategy

この呼び出し元のデフォルトの認可方法 (ALLOWまたはDENY)。

issuer

トークン発行者。

sourceIp

が AWS AppSync 受信する発信者の送信元 IP アドレス。リクエストに `x-forwarded-for` ヘッダーが含まれていない場合、ソース IP の値には TCP 接続からの 1 つの IP アドレスのみが含まれます。リクエストに `x-forwarded-for` ヘッダーが含まれている場合、ソース IP は、TCP 接続からの IP アドレスと `x-forwarded-for` ヘッダーからの IP アドレスのリストです。

sub

認証されたユーザーの UUID。

user

IAM ユーザー。

userArn

IAM ユーザーの Amazon リソースネーム (ARN)。

username

認証されたユーザーのユーザー名です。AMAZON_COGNITO_USER_POOLS 認可の場合、username の値は cognito:username 属性の値です。AWS_IAM 認可の場合、ユーザー名の値は AWS ユーザープリンシパルの値です。Amazon Cognito アイデンティティプールから発行された認証情報による IAM 認可を使用されている場合は、cognitoIdentityIdの使用をお勧めします。

リクエストヘッダーへのアクセス

AWS AppSync は、クライアントからのカスタムヘッダーの受け渡しと、 を使用した GraphQL リゾルバーでのカスタムヘッダーへのアクセスをサポートしています$context.request.headers。データソースへデータの挿入や認可チェックなどのアクションでそのヘッダーの値を使用できます。次の例のようにコマンドラインから API キーで $curl を使用して、1 つまたは複数のリクエストヘッダーを使用できます。

単一ヘッダーの例

次のように、custom のヘッダーに nadia の値を設定しているとします。

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:nadia" -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}"}' https://<ENDPOINT>/graphql
```

この値には $context.request.headers.custom を使用してアクセスできます。たとえば、DynamoDB に対する VTL は次のようになります。

```
"custom": $util.dynamodb.toDynamoDBJson($context.request.headers.custom)
```

複数ヘッダーの例

1 つのリクエストで複数のヘッダーを渡して、リゾルバーのマッピングテンプレートでそれらのヘッダーにアクセスすることもできます。例えば、次のように custom ヘッダーに 2 つの値が設定されているとします。

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:bailey" -H "custom:nadia"
 -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo
\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}"}'
 https://<ENDPOINT>/graphql
```

それらのヘッダーに配列としてアクセスできます (例: `$context.request.headers.custom[1]`)。

> **Note**
>
> AWS AppSync は の Cookie ヘッダーを公開しません`$context.request.headers`。

リクエストカスタムドメイン名にアクセスする

AWS AppSync は、APIs の GraphQL およびリアルタイムエンドポイントにアクセスするために使用できるカスタムドメインの設定をサポートしています。カスタムドメイン名を使用してリクエストを行う場合は、`$context.request.domainName` を使用してドメイン名を取得できます。

デフォルトの GraphQL エンドポイントドメイン名を使用する場合、値は `null` です。

情報

`info` セクションには、GraphQL リクエストに関する情報が含まれています。このセクションには、次の形式が含まれます。

```
{
    "fieldName": "string",
    "parentTypeName": "string",
    "variables": { ... },
    "selectionSetList": ["string"],
    "selectionSetGraphQL": "string"
}
```

各フィールドは次のように定義されています。

`fieldName`

現在解決中のフィールドの名前。

parentTypeName

現在解決中のフィールドの親タイプの名前。

variables

GraphQLリクエストに渡されるすべての変数を保持するマップ。

selectionSetList

GraphQL 選択セット内のフィールドのリスト表現。エイリアスされたフィールドは、フィールド名ではなく、エイリアス名によってのみ参照されます。次の例で詳細を示します。

selectionSetGraphQL

選択セットの文字列表現。GraphQL スキーマ定義言語 (SDL) としてフォーマットされます。フラグメントは選択セットにマージされませんが、次の例に示すように、インラインフラグメントは保持されます。

> **Note**
>
> `context.info` を `$utils.toJson()` で使用する場合 , `selectionSetList` および `selectionSetGraphQL` によって返される値はデフォルトでシリアル化されません。

たとえば、次のクエリの `getPost` フィールドを解決する場合:

```
query {
  getPost(id: $postId) {
    postId
    title
    secondTitle: title
    content
    author(id: $authorId) {
      authorId
      name
    }
    secondAuthor(id: "789") {
      authorId
    }
    ... on Post {
      inlineFrag: comments: {
```

```
        id
      }
    }
    ... postFrag
  }
}

fragment postFrag on Post {
  postFrag: comments: {
    id
  }
}
```

レスポンスマッピングテンプレートを処理する際に使用可能な完全な $context.info 変数は次のようになります。

```
{
  "fieldName": "getPost",
  "parentTypeName": "Query",
  "variables": {
    "postId": "123",
    "authorId": "456"
  },
  "selectionSetList": [
    "postId",
    "title",
    "secondTitle"
    "content",
    "author",
    "author/authorId",
    "author/name",
    "secondAuthor",
    "secondAuthor/authorId",
    "inlineFragComments",
    "inlineFragComments/id",
    "postFragComments",
    "postFragComments/id"
  ],
  "selectionSetGraphQL": "{\n  getPost(id: $postId) {\n    postId\n    title\n
 secondTitle: title\n    content\n    author(id: $authorId) {\n      authorId\n
 name\n    }\n    secondAuthor(id: \"789\") {\n      authorId\n    }\n    ... on Post
 {\n      inlineFrag: comments {\n        id\n      }\n    }\n    ... postFrag\n  }\n}"
}
```

selectionSetList は現在のタイプに属するフィールドのみを公開します。現在のタイプがインタフェースまたは共用体の場合、そのインタフェースに属する選択されたフィールドのみが公開されます。例として、次のスキーマがあるとします。

```
type Query {
    node(id: ID!): Node
}

interface Node {
    id: ID
}

type Post implements Node {
    id: ID
    title: String
    author: String
}

type Blog implements Node {
    id: ID
    title: String
    category: String
}
```

そして次のクエリがあります。

```
query {
    node(id: "post1") {
        id
        ... on Post {
            title
        }

        ... on Blog {
            title
        }
    }
}
```

Query.node フィールド解決で $ctx.info.selectionSetList が呼び出されると、id のみが公開されます。

```
"selectionSetList": [
    "id"
]
```

入力のサニタイズ

アプリケーションは、信頼できない入力をサニタイズして、部外者が意図した用途以外でアプリケーションを使用することを防ぐ必要があります。$context には、$context.arguments、$context.identity、$context.result、$context.info.variables などのプロパティにユーザー入力が含まれているため、マッピングテンプレート内の値をサニタイズするように注意する必要があります。

マッピングテンプレートは JSON を表すため、入力のサニタイズは、ユーザー入力を表す文字列から JSON 予約文字をエスケープする形式をとります。JSON 予約文字をマッピングテンプレートに配置するときは、$util.toJson() ユーティリティを使用して機密文字列値からエスケープすることをお勧めします。

たとえば、以下の Lambda リクエストマッピングテンプレートでは、安全でないユーザー入力文字列 ($context.arguments.id) にアクセスしたため、エスケープされていない JSON 文字が JSON テンプレートを破壊するのを防ぐために、$util.toJson() でラップしました。

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": $util.toJson($context.arguments.id)
    }
}
```

$context.arguments.id をサニタイズなしで直接挿入する以下のマッピングテンプレートとは対照的です。これは、エスケープされていない二重引用符やその他の JSON 予約文字を含む文字列では機能せず、テンプレートを開くとエラーになる可能性があります。

```
## DO NOT DO THIS
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
```

```
        "field": "getPost",
        "postId": "$context.arguments.id" ## Unsafe! Do not insert $context string
 values without escaping JSON characters.
    }
}
```

AWS AppSync リゾルバーマッピングテンプレートユーティリティリファレンス

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync は、データソースとのやり取りを簡素化するために GraphQL リゾルバー内で使用できる一連のユーティリティを定義します。これらのユーティリティの中には、ID やタイムスタンプの生成など、あらゆるデータソースで一般的に使用されるものもあります。その他には、データソースの種類に固有のものもあります。次のユーティリティを使用できます。

- $util のユーティリティヘルパー - $util変数には、データの操作を容易にする一般的なユーティリティメソッドが含まれています。特に指定されていない限り、すべてのユーティリティでは UTF-8 文字セットが使用されます。
- AppSync ディレクティブ - AppSync では、VTL の記述時に開発者の生産性を高めるディレクティブが公開されています。
- $util.time のタイムヘルパー - $util.time 変数には、タイムスタンプの生成、日時形式間の変換、および日時文字列の解析に役立つ日時メソッドが含まれています。日時形式の構文は DateTimeFormatter に基づいています。詳細については、DateTimeFormatter を参照してください。
- $util.list のリストヘルパー - $util.list には、よく使用されるリストオペレーション (フィルタリングのユースケースでのリストの項目の削除や保持など) に役立つメソッドが含まれています。
- $util.map のマップヘルパー - $util.map には、よく使用されるマップオペレーション (フィルタリングのユースケースでのマップの項目の削除や保持など) に役立つメソッドが含まれています。
- $util.dynamodb のDynamoDB ヘルパー - $util.dynamodb には、Amazon DynamoDB に対するデータの読み書きを容易にするヘルパーメソッド (自動型マッピングやフォーマットなど) が含まれています。

- $util.rds の Amazon RDS ヘルパー - $util.rds には、結果出力から不要なデータを削除して RDS オペレーションをフォーマットするヘルパーメソッドが含まれています。
- $util.http の HTTP ヘルパー - $util.http ユーティリティには、HTTP リクエストパラメータの管理やレスポンスヘッダーの追加に使用できるヘルパーメソッドが用意されています。
- $util.xml の XML ヘルパー - $util.xml には、XML レスポンスを JSON またはディクショナリに変換しやすくするヘルパーメソッドが含まれています。
- $util.transform の変換ヘルパー - $util.transform には、DynamoDB フィルターオペレーションのような、データソースに対して複雑な操作を簡単に実行できるヘルパーメソッドが含まれています。
- $util.math の数学ヘルパー - $util.math には、一般的な数学オペレーションに役立つメソッドが含まれています。
- $util.str の文字列ヘルパー - $util.str には、一般的な文字列オペレーションに役立つメソッドが含まれています。
- 拡張機能 - $extensions には、リゾルバー内で追加のアクションを行うためのメソッドのセットが含まれています。

$util のユーティリティヘルパー

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちらにある APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

`$util` 変数には、データの操作を容易にする一般的なユーティリティメソッドが含まれています。特に指定されていない限り、すべてのユーティリティでは UTF-8 文字セットが使用されます。

JSON パーシング utils

JSON パーシング utils リスト

`$util.parseJson(String) : Object`

"stringified" JSON を受け取り、結果のオブジェクト表現を返します。

`$util.toJson(Object) : String`

オブジェクトを受け取り、そのオブジェクトの「文字列化された」JSON 表現を返します。

エンコーディング utils

エンコーディング utils リスト

`$util.urlEncode(String) : String`

入力文字列を `application/x-www-form-urlencoded` でエンコードされた文字列として返します。

`$util.urlDecode(String) : String`

`application/x-www-form-urlencoded` でエンコードされた文字列をデコードして、エンコードされていない形式に戻します。

`$util.base64Encode( byte[] ) : String`

入力を base64 でエンコードされた文字列にエンコードします。

`$util.base64Decode(String) : byte[]`

base64 エンコードされた文字列からデータをデコードします。

ID 生成 utils

ID 生成 utils リスト

`$util.autoId() : String`

ランダムに生成された 128 ビットの UUID を返します。

`$util.autoUlid() : String`

ランダムに生成された 128 ビットの ULID (辞書的にソート可能なユニバーサルユニーク識別子) を返します。

`$util.autoKsuid() : String`

長さ 27 の文字列としてエンコードされた、ランダムに生成された 128 ビットの KSUID (K ソート可能な固有識別子) base62 を返します。

エラー utils

エラー utils リスト

`$util.error(String)`

カスタムエラーをスローします。リクエストや呼び出し結果のエラーを検出するために、リクエストやレスポンスのマッピングテンプレートでこれを使用します。

`$util.error(String, String)`

カスタムエラーをスローします。リクエストや呼び出し結果のエラーを検出するために、リクエストやレスポンスのマッピングテンプレートでこれを使用します。`errorType` を指定することもできます。

`$util.error(String, String, Object)`

カスタムエラーをスローします。リクエストや呼び出し結果のエラーを検出するために、リクエストやレスポンスのマッピングテンプレートでこれを使用します。`errorType` と `data` フィールドを指定することもできます。`data` の値は、GraphQL レスポンスの `errors` 内の該当する `error` ブロックに追加されます。

> **Note**
>
> `data` はクエリ選択セットに基づいてフィルタリングされます。

`$util.error(String, String, Object, Object)`

カスタムエラーをスローします。これは、テンプレートがリクエストまたは呼び出し結果でエラーを検出した場合に、リクエストまたはレスポンスのマッピングテンプレートで使用できます。また、`errorType` フィールド、`data` フィールド、および `errorInfo` フィールドを指定できます。`data` の値は、GraphQL レスポンスの `errors` 内の該当する `error` ブロックに追加されます。

> **Note**
>
> `data` はクエリ選択セットに基づいてフィルタリングされます。`errorInfo` の値は、GraphQL レスポンスの `errors` 内の該当する `error` ブロックに追加されます。`errorInfo` はクエリ選択セットに基づいてフィルタリングされません。

$util.appendError(String)

カスタムエラーを追加します。これは、テンプレートがリクエストまたは呼び出し結果でエラーを検出した場合に、リクエストまたはレスポンスのマッピングテンプレートで使用できます。$util.error(String) とは異なり、テンプレートの評価は中断されないため、データを呼び出し元に返すことができます。

$util.appendError(String, String)

カスタムエラーを追加します。これは、テンプレートがリクエストまたは呼び出し結果でエラーを検出した場合に、リクエストまたはレスポンスのマッピングテンプレートで使用できます。また、errorType を指定できます。$util.error(String, String) とは異なり、テンプレートの評価は中断されないため、データを呼び出し元に返すことができます。

$util.appendError(String, String, Object)

カスタムエラーを追加します。これは、テンプレートがリクエストまたは呼び出し結果でエラーを検出した場合に、リクエストまたはレスポンスのマッピングテンプレートで使用できます。また、errorType と data フィールドを指定できます。$util.error(String, String, Object) とは異なり、テンプレートの評価は中断されないため、データを呼び出し元に返すことができます。data の値は、GraphQL レスポンスの errors 内の該当する error ブロックに追加されます。

> **Note**
>
> data はクエリ選択セットに基づいてフィルタリングされます。

$util.appendError(String, String, Object, Object)

カスタムエラーを追加します。これは、テンプレートがリクエストまたは呼び出し結果でエラーを検出した場合に、リクエストまたはレスポンスのマッピングテンプレートで使用できます。また、errorType フィールド、data フィールド、および errorInfo フィールドを指定できます。$util.error(String, String, Object, Object) とは異なり、テンプレートの評価は中断されないため、データを呼び出し元に返すことができます。data の値は、GraphQL レスポンスの errors 内の該当する error ブロックに追加されます。

> **Note**
>
> data はクエリ選択セットに基づいてフィルタリングされます。errorInfo の値は、GraphQL レスポンスの errors 内の該当する error ブロックに追加されます。

errorInfo はクエリ選択セットに基づいてフィルタリングされません。

条件検証 utils

条件検証 utils リスト

$util.validate(Boolean, String) : void

条件が false の場合、指定されたメッセージ付きで CustomTemplateException がスローされます。

$util.validate(Boolean, String, String) : void

条件が false の場合、指定されたメッセージとエラータイプ付きで CustomTemplateException がスローされます。

$util.validate(Boolean, String, String, Object) : void

条件が false の場合、指定されたメッセージ、エラータイプ、およびレスポンスで返されたデータ付きで CustomTemplateException がスローされます。

Null 動作 utils

Null 動作 utils リスト

$util.isNull(Object) : Boolean

指定されたオブジェクトが null である場合は true を返します。

$util.isNullOrEmpty(String) : Boolean

指定されたデータが null または空の文字列である場合は true を返します。それ以外の場合は、false を返します。

$util.isNullOrBlank(String) : Boolean

指定されたデータが null または空白文字列である場合は true を返します。それ以外の場合は、false を返します。

$util.defaultIfNull(Object, Object) : Object

最初のオブジェクトが null でない場合は、そのオブジェクトを返します。それ以外の場合は、2 番目のオブジェクトを「デフォルトオブジェクト」として返します。

`$util.defaultIfNullOrEmpty(String, String) : String`

最初の文字列型が null でも空の文字列でもない場合は、その文字列を返します。それ以外の場合は、2 番目の文字列型を「デフォルト文字列」として返します。

`$util.defaultIfNullOrBlank(String, String) : String`

最初の文字列型が null でも空白文字列でもない場合は、その文字列型を返します。それ以外の場合は、2 番目の文字列型を「デフォルト文字列」として返します。

パターンのマッチング utils

タイプとパターンの一致 utils リスト

`$util.typeOf(Object) : String`

オブジェクトの型を表す文字列型を返します。サポートされている型識別名は "Null"、"Number"、"String"、"Map"、"List"、"Boolean" です。型を識別できない場合は、"Object" を返します。

`$util.matches(String, String) : Boolean`

最初の引数で指定されたパターンが、2 番目の引数で指定されたデータと一致する場合は true を返します。パターンは正規表現である必要があります (例: `$util.matches("a*b", "aaaaab")`)。この機能は Java の Pattern に基づいています。詳細については、Pattern を参照してください。

`$util.authType() : String`

リクエストで使用されているマルチ認可タイプを表す文字列型 (String) の値を返します。「IAM 認可」、「ユーザープールの認可」、「オープン ID Connect 認可」、または「API キー認可」のいずれかを返します。

Object validation utils

Object validation utils リスト

`$util.isString(Object) : Boolean`

オブジェクトが文字列型である場合は true を返します。

$util.isNumber(Object) : Boolean

オブジェクトが数値型である場合は true を返します。

$util.isBoolean(Object) : Boolean

オブジェクトがブール型である場合は true を返します。

$util.isList(Object) : Boolean

オブジェクトがリスト型である場合は true を返します。

$util.isMap(Object) : Boolean

オブジェクトがマップ型である場合は true を返します。

CloudWatch ログ記録 Utils

CloudWatch ログ記録 Utils リスト

$util.log.info(Object) : Void

API でログレベル 、ALL、または でリクエストレベルおよびフィールドレベルの CloudWatch ログ記録が有効になっている場合INFOに、指定されたオブジェクトの文字列表現をリクエストされたログストリームDEBUGに記録します。

$util.log.info(String, Object...) : Void

API のログレベル ALL でリクエストレベルとフィールドレベルの CloudWatch ロギングが有効になっている場合、提供されたオブジェクトの String 表現をリクエストされたログストリームに記録します。このユーティリティは、最初の入力フォーマット String の「{}」で示されるすべての変数を、指定されたオブジェクトの文字列表現に順番に置き換えます。

$util.log.debug(Object) : Void

ログレベルALLまたは API でリクエストレベルおよびフィールドレベルの CloudWatch ログ記録が有効になっている場合、指定されたオブジェクトの文字列表現をリクエストされたログストリームDEBUGに記録します。

$util.log.debug(String, Object...) : Void

ログレベル DEBUG または API のログレベル ALL でフィールドレベルの CloudWatch ロギングが有効になっている場合、要求されたログストリームに提供されたオブジェクトの文字列表現を記録します。このユーティリティは、最初の入力フォーマット String の「{}」で示されるすべての変数を、指定されたオブジェクトの文字列表現に順番に置き換えます。

$util.log.error(Object) : Void

API の任意のログレベル (ALL、、 DEBUGなど) でフィールドレベルの CloudWatch ログ記録が有効になっている場合INFO、指定されたオブジェクトの文字列表現をリクエストされたログストリームに記録します。

$util.log.error(String, Object...) : Void

ログレベル ERROR または API のログレベル ALL でフィールドレベルの CloudWatch ロギングが有効になっている場合、要求されたログストリームに提供されたオブジェクトの文字列表現を記録します。このユーティリティは、最初の入力フォーマット String の「{}」で示されるすべての変数を、指定されたオブジェクトの文字列表現に順番に置き換えます。

戻り値の動作 utils

戻り値の動作 utils リスト

$util.qr() および **$util.quiet()**

VTL ステートメントを実行し、返される値を抑えます。これは、マップへの項目の追加など、一時的なプレースホルダーを使用せずにメソッドを実行する場合に便利です。例:

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

次のようになります。

```
#set ($myMap = {})
$util.qr($myMap.put("id", "first value"))
```

$util.escapeJavaScript(String) : String

入力文字列を JavasScript のエスケープした文字列として返します。

$util.urlEncode(String) : String

入力文字列を application/x-www-form-urlencoded でエンコードされた文字列として返します。

$util.urlDecode(String) : String

application/x-www-form-urlencoded でエンコードされた文字列をデコードして、エンコードされていない形式に戻します。

$util.base64Encode(byte[]) : String

入力を base64 でエンコードされた文字列にエンコードします。

$util.base64Decode(String) : byte[]

base64 エンコードされた文字列からデータをデコードします。

$util.parseJson(String) : Object

"stringified" JSON を受け取り、結果のオブジェクト表現を返します。

$util.toJson(Object) : String

オブジェクトを受け取り、そのオブジェクトの「文字列化された」JSON 表現を返します。

$util.autoId() : String

ランダムに生成された 128 ビットの UUID を返します。

$util.autoUlid() : String

ランダムに生成された 128 ビットの ULID (辞書的にソート可能なユニバーサルユニーク識別子) を返します。

$util.autoKsuid() : String

長さ 27 の文字列としてエンコードされた、ランダムに生成された 128 ビットの KSUID (K ソート可能な固有識別子) base62 を返します。

$util.unauthorized()

解決されるフィールドに対して `Unauthorized` をスローします。これは、呼び出し元にそのフィールドの解決が許可されるかどうかを判別するために、リクエストまたはレスポンスのマッピングテンプレートで使用できます。

$util.error(String)

カスタムエラーをスローします。リクエストや呼び出し結果のエラーを検出するために、リクエストやレスポンスのマッピングテンプレートでこれを使用します。

$util.error(String, String)

カスタムエラーをスローします。リクエストや呼び出し結果のエラーを検出するために、リクエストやレスポンスのマッピングテンプレートでこれを使用します。`errorType` を指定することもできます。

$util.error(String, String, Object)

カスタムエラーをスローします。リクエストや呼び出し結果のエラーを検出するために、リクエストやレスポンスのマッピングテンプレートでこれを使用します。errorType と data フィールドを指定することもできます。data の値は、GraphQL レスポンスの errors 内の該当する error ブロックに追加されます。注 : data はクエリ選択セットに基づいてフィルタリングされます。

$util.error(String, String, Object, Object)

カスタムエラーをスローします。これは、テンプレートがリクエストまたは呼び出し結果でエラーを検出した場合に、リクエストまたはレスポンスのマッピングテンプレートで使用できます。また、errorType フィールド、data フィールド、および errorInfo フィールドを指定できます。data の値は、GraphQL レスポンスの errors 内の該当する error ブロックに追加されます。注 : data はクエリ選択セットに基づいてフィルタリングされます。errorInfo の値は、GraphQL レスポンスの errors 内の該当する error ブロックに追加されます。注 : errorInfo はクエリ選択セットに基づいてフィルタリングされません。

$util.appendError(String)

カスタムエラーを追加します。これは、テンプレートがリクエストまたは呼び出し結果でエラーを検出した場合に、リクエストまたはレスポンスのマッピングテンプレートで使用できます。$util.error(String) とは異なり、テンプレートの評価は中断されないため、データを呼び出し元に返すことができます。

$util.appendError(String, String)

カスタムエラーを追加します。これは、テンプレートがリクエストまたは呼び出し結果でエラーを検出した場合に、リクエストまたはレスポンスのマッピングテンプレートで使用できます。また、errorType を指定できます。$util.error(String, String) とは異なり、テンプレートの評価は中断されないため、データを呼び出し元に返すことができます。

$util.appendError(String, String, Object)

カスタムエラーを追加します。これは、テンプレートがリクエストまたは呼び出し結果でエラーを検出した場合に、リクエストまたはレスポンスのマッピングテンプレートで使用できます。また、errorType と data フィールドを指定できます。$util.error(String, String, Object) とは異なり、テンプレートの評価は中断されないため、データを呼び出し元に返すことができます。data の値は、GraphQL レスポンスの errors 内の該当する error ブロックに追加されます。注 : data はクエリ選択セットに基づいてフィルタリングされます。

$util.appendError(String, String, Object, Object)

カスタムエラーを追加します。これは、テンプレートがリクエストまたは呼び出し結果でエラーを検出した場合に、リクエストまたはレスポンスのマッピングテンプレートで使用できます。また、errorType フィールド、data フィールド、および errorInfo フィールドを指定できます。$util.error(String, String, Object, Object) とは異なり、テンプレートの評価は中断されないため、データを呼び出し元に返すことができます。data の値は、GraphQL レスポンスの errors 内の該当する error ブロックに追加されます。注 : data はクエリ選択セットに基づいてフィルタリングされます。errorInfo の値は、GraphQL レスポンスの errors 内の該当する error ブロックに追加されます。注 : errorInfo はクエリ選択セットに基づいてフィルタリングされません。

$util.validate(Boolean, String) : void

条件が false の場合、指定されたメッセージ付きで CustomTemplateException がスローされます。

$util.validate(Boolean, String, String) : void

条件が false の場合、指定されたメッセージとエラータイプ付きで CustomTemplateException がスローされます。

$util.validate(Boolean, String, String, Object) : void

条件が false の場合、指定されたメッセージ、エラータイプ、およびレスポンスで返されたデータ付きで CustomTemplateException がスローされます。

$util.isNull(Object) : Boolean

指定されたオブジェクトが null である場合は true を返します。

$util.isNullOrEmpty(String) : Boolean

指定されたデータが null または空の文字列である場合は true を返します。それ以外の場合は、false を返します。

$util.isNullOrBlank(String) : Boolean

指定されたデータが null または空白文字列である場合は true を返します。それ以外の場合は、false を返します。

$util.defaultIfNull(Object, Object) : Object

最初のオブジェクトが null でない場合は、そのオブジェクトを返します。それ以外の場合は、2 番目のオブジェクトを「デフォルトオブジェクト」として返します。

$util.defaultIfNullOrEmpty(String, String) : String

最初の文字列型が null でも空の文字列でもない場合は、その文字列を返します。それ以外の場合は、2 番目の文字列型を「デフォルト文字列」として返します。

$util.defaultIfNullOrBlank(String, String) : String

最初の文字列型が null でも空白文字列でもない場合は、その文字列型を返します。それ以外の場合は、2 番目の文字列型を「デフォルト文字列」として返します。

$util.isString(Object) : Boolean

オブジェクトが文字列型である場合は true を返します。

$util.isNumber(Object) : Boolean

オブジェクトが数値型である場合は true を返します。

$util.isBoolean(Object) : Boolean

オブジェクトがブール型である場合は true を返します。

$util.isList(Object) : Boolean

オブジェクトがリスト型である場合は true を返します。

$util.isMap(Object) : Boolean

オブジェクトがマップ型である場合は true を返します。

$util.typeOf(Object) : String

オブジェクトの型を表す文字列型を返します。サポートされている型識別名は "Null"、"Number"、"String"、"Map"、"List"、"Boolean" です。型を識別できない場合は、"Object" を返します。

$util.matches(String, String) : Boolean

最初の引数で指定されたパターンが、2 番目の引数で指定されたデータと一致する場合は true を返します。パターンは正規表現である必要があります (例: `$util.matches("a*b", "aaaaab")`)。この機能は Java の Pattern に基づいています。詳細については、Pattern を参照してください。

$util.authType() : String

リクエストで使用されているマルチ認可タイプを表す文字列型 (String) の値を返します。「IAM 認可」、「ユーザープールの認可」、「オープン ID Connect 認可」、または「API キー認可」のいずれかを返します。

$util.log.info(Object) : Void

API のログレベル ALL でリクエストレベルとフィールドレベルの CloudWatch ロギングが有効になっている場合、提供されたオブジェクトの String 表現をリクエストされたログストリームに記録します。

$util.log.info(String, Object...) : Void

API のログレベル ALL でリクエストレベルとフィールドレベルの CloudWatch ロギングが有効になっている場合、提供されたオブジェクトの String 表現をリクエストされたログストリームに記録します。このユーティリティは、最初の入力フォーマット String の「{}」で示されるすべての変数を、指定されたオブジェクトの文字列表現に順番に置き換えます。

$util.log.error(Object) : Void

ログレベル ERROR または API のログレベル ALL でフィールドレベルの CloudWatch ロギングが有効になっている場合、要求されたログストリームに提供されたオブジェクトの文字列表現を記録します。

$util.log.error(String, Object...) : Void

ログレベル ERROR または API のログレベル ALL でフィールドレベルの CloudWatch ロギングが有効になっている場合、要求されたログストリームに提供されたオブジェクトの文字列表現を記録します。このユーティリティは、最初の入力フォーマット String の「{}」で示されるすべての変数を、指定されたオブジェクトの文字列表現に順番に置き換えます。

$util.escapeJavaScript(String) : String

入力文字列を JavasScript のエスケープした文字列として返します。

リゾルバー認可

リゾルバー承認リスト

$util.unauthorized()

解決されるフィールドに対して Unauthorized をスローします。これは、呼び出し元にそのフィールドの解決が許可されるかどうかを判別するために、リクエストまたはレスポンスのマッピングテンプレートで使用できます。

AWS AppSync ディレクティブ

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync は、VTL で記述するときに開発者の生産性を向上させるためのディレクティブを公開します。

ディレクティブ utils

`#return(Object)`

`#return(Object)` により、どのマッピングテンプレートからも早期に戻ることができます。`#return(Object)` はプログラミング言語の return キーワードに似ており、最も近いスコープのロジックブロックから戻ります。つまり、リゾルバーのマッピングテンプレート内で `#return(Object)` を使用すると、リゾルバーから戻ることになります。さらに、関数のマッピングテンプレートから `#return(Object)` を使用すると、実行は関数から戻り、パイプライン内の次の関数に、またはリゾルバーのレスポンスマッピングテンプレートに継続されます。

`#return`

`#return` ディレクティブは `#return(Object)` と同じ動作をしますが、`null` は代わりに返されます。

$util.time の日時ヘルパー

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちらにある APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

`$util.time` 変数には、タイムスタンプの生成、日時形式間の変換、および日時文字列の解析に役立つ日時メソッドが含まれています。日時形式の構文は Java の DateTimeFormatter に基づいています。詳細については、DateTimeFormatter を参照してください。いくつかの例、および利用可能なメソッドの一覧と説明を以下に示します。

日時 utils

日時 utils リスト

$util.time.nowISO8601() : String

ISO 8601 形式の UTC の文字列型表現を返します。

$util.time.nowEpochSeconds() : long

エポック (1970-01-01T00:00:00Z) から現在までの秒数を返します。

$util.time.nowEpochMilliSeconds() : long

エポック (1970-01-01T00:00:00Z) から現在までのミリ秒数を返します。

$util.time.nowFormatted(String) : String

文字列型の入力引数で指定された形式を使用して、UTC での現在のタイムスタンプを返します。

$util.time.nowFormatted(String, String) : String

文字列型の入力引数で指定された形式とタイムゾーンを使用して、そのタイムゾーンでの現在のタイムスタンプを返します。

$util.time.parseFormattedToEpochMilliSeconds(String, String) : Long

文字列として渡されたタイムスタンプとタイムゾーンを含む形式を解析し、エポックからのミリ秒単位でタイムスタンプを返します。

$util.time.parseFormattedToEpochMilliSeconds(String, String, String) : Long

文字列型として渡されたタイムスタンプ、形式、およびタイムゾーンを解析し、エポックからのミリ秒単位のタイムスタンプを返します。

$util.time.parseISO8601ToEpochMilliSeconds(String) : Long

文字列型として渡された ISO8601 形式のタイムスタンプを解析し、エポックからのミリ秒単位のタイムスタンプを返します。

$util.time.epochMilliSecondsToSeconds(long) : long

エポックからのミリ秒単位のタイムスタンプを、エポックからの秒単位のタイムスタンプに変換します。

$util.time.epochMilliSecondsToISO8601(long) : String

エポックからのミリ秒単位のタイムスタンプを、ISO8601 形式のタイムスタンプに変換します。

$util.time.epochMilliSecondsToFormatted(long, String) : String

long として渡されたエポックからのミリ秒単位のタイムスタンプを、文字列型で指定された形式に合わせて UTC でのタイムスタンプに変換します。

$util.time.epochMilliSecondsToFormatted(long, String, String) : String

long として渡されたエポックからのミリ秒単位のタイムスタンプを、文字列型で指定された形式とタイムゾーンに合わせて、そのタイムゾーンでのタイムスタンプに変換します。

スタンドアロン関数の例

```
$util.time.nowISO8601()                                              :
 2018-02-06T19:01:35.749Z
$util.time.nowEpochSeconds()                                         : 1517943695
$util.time.nowEpochMilliSeconds()                                    : 1517943695750
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ")                      : 2018-02-06
 19:01:35+0000
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ", "+08:00")            : 2018-02-07
 03:01:35+0800
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ", "Australia/Perth") : 2018-02-07
 03:01:35+0800
```

変換の例

```
#set( $nowEpochMillis = 1517943695758 )
$util.time.epochMilliSecondsToSeconds($nowEpochMillis)
    : 1517943695
$util.time.epochMilliSecondsToISO8601($nowEpochMillis)
    : 2018-02-06T19:01:35.758Z
$util.time.epochMilliSecondsToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ")
    : 2018-02-06 19:01:35+0000
$util.time.epochMilliSecondsToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ",
 "+08:00") : 2018-02-07 03:01:35+0800
```

解析の例

```
$util.time.parseISO8601ToEpochMilliSeconds("2018-02-01T17:21:05.180+08:00")
              : 1517476865180
$util.time.parseFormattedToEpochMilliSeconds("2018-02-02 01:19:22+0800", "yyyy-MM-dd
 HH:mm:ssZ")     : 1517505562000
```

```
$util.time.parseFormattedToEpochMilliSeconds("2018-02-02 01:19:22", "yyyy-MM-dd
 HH:mm:ss", "+08:00") : 1517505562000
```

AWS AppSync 定義されたスカラーの使用

次の形式は、AWSDate、AWSDateTime、および AWSTime と互換性があります。

```
$util.time.nowFormatted("yyyy-MM-dd[XXX]", "-07:00:30")              :
 2018-07-11-07:00
$util.time.nowFormatted("yyyy-MM-dd'T'HH:mm:ss[XXXXX]", "-07:00:30")  :
 2018-07-11T15:14:15-07:00:30
```

$util.list のヘルパーのリスト

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちらにある APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

`$util.list` には、よく使用されるリストオペレーション (フィルタリングのユースケースでのリストの項目の削除や保持など) に役立つメソッドが含まれています。

utils を一覧表示する

`$util.list.copyAndRetainAll(List, List) : List`

最初の引数で指定されたリストのシャローコピーを作成し、2 番目の引数で指定された項目のみを保持します (存在する場合)。他のすべての項目はそのコピーから削除されます。

`$util.list.copyAndRemoveAll(List, List) : List`

最初の引数で指定されたリストのシャローコピーを作成し、2 番目の引数で指定されたすべての項目を削除します (存在する場合)。他のすべての項目はそのコピーで保持されます。

`$util.list.sortList(List, Boolean, String) : List`

最初の引数で指定されたオブジェクトのリストをソートします。2 番目の引数が true の場合、リストは降順でソートされます。2 番目の引数が false の場合、リストは昇順でソートされます。3 番目の引数は、カスタムオブジェクトのリストをソートするために使用されるプロパティの文字列名です。文字列、整数、浮動小数点数、倍精度浮動小数点数だけのリストの場合、3 番目の引数は任意のランダムな文字列になります。すべてのオブジェクトが同じクラスのものでない場合

は、元のリストが返されます。最大 1000 個のオブジェクトを含むリストのみがサポートされます。以下は、このユーティリティの使用例です。

```
 INPUT:      $util.list.sortList([{"description":"youngest", "age":5},
{"description":"middle", "age":45}, {"description":"oldest", "age":85}], false,
 "description")
 OUTPUT:     [{"description":"middle", "age":45}, {"description":"oldest",
 "age":85}, {"description":"youngest", "age":5}]
```

$util.map のマップヘルパー

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちらにある APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

`$util.map` には、よく使用されるマップオペレーション (フィルタリングのユースケースでのマップの項目の削除や保持など) に役立つメソッドが含まれています。

Map utils

`$util.map.copyAndRetainAllKeys(Map, List) : Map`

最初の引数で指定されたマップのシャローコピーを作成し、リストで指定されたキーのみを保持します (存在する場合)。他のすべてのキーはそのコピーから削除されます。

`$util.map.copyAndRemoveAllKeys(Map, List) : Map`

最初の引数で指定されたマップのシャローコピーを作成し、リストで指定されたキーを持つすべてのエントリを削除します (存在する場合)。他のすべてのキーはそのコピーで保持されます。

$util.dynamodb の DynamoDB ヘルパー

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちらにある APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

$util.dynamodb には、Amazon DynamoDB に対するデータの読み書きを容易にするヘルパーメソッド (自動型マッピングやフォーマットなど) が含まれています。これらのメソッドは、プリミティブ型やリスト型を適切な DynamoDB 入力形式に自動的にマッピングするように設計されていて、そのマッピングは { "TYPE" : VALUE } 形式の Map です。

例えば、前述の例で、DynamoDB に新しい項目を作成するリクエストマッピングテンプレートは次のようになっていました。

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "id" : { "S" : "$util.autoId()" }
    },
    "attributeValues" : {
         "title" : { "S" : $util.toJson($ctx.args.title) },
         "author" : { "S" : $util.toJson($ctx.args.author) },
         "version" : { "N", $util.toJson($ctx.args.version) }
    }
}
```

このオブジェクトにフィールドを追加する場合は、スキーマで GraphQL でクエリを更新し、リクエストマッピングテンプレートも更新する必要があります。ただし、スキーマに追加された新しいフィールドが自動的に取得され、正しい型で DynamoDB に追加されるように、リクエストマッピングテンプレートを再構築できるようになりました。

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

上記の例では、$util.dynamodb.toDynamoDBJson(...) ヘルパーを使用して、生成された ID を自動的に取得し、それを文字列属性の DynamoDB 表現に変換しています。次に、すべての引数を取得して DynamoDB 表現に変換し、それをテンプレートの attributeValues フィールドに出力しています。

各ヘルパーには、オブジェクトを返すバージョン (例 : $util.dynamodb.toString(...)) と、オブジェクトを JSON 文字列として返すバージョン (例 : $util.dynamodb.toStringJson(...)) の 2 つのバージョンがあります。上記の例では、データを JSON 文字列として返すバージョンを使用していました。次に示すように、オブジェクトがテンプレートで使用される前にそのオブジェクトを操作する場合は、代わりにオブジェクトを返すことを選択できます。

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
    },

    #set( $myFoo = $util.dynamodb.toMapValues($ctx.args) )
    #set( $myFoo.version = $util.dynamodb.toNumber(1) )
    #set( $myFoo.timestamp = $util.dynamodb.toString($util.time.nowISO8601()))

    "attributeValues" : $util.toJson($myFoo)
}
```

前述の例では、変換された引数を JSON 文字列としてではなくマップとして返し、version フィールドと timestamp フィールドを追加してから、最終的にテンプレートで attributeValues を使用してそれらを $util.toJson(...) フィールドに出力していました。

各ヘルパーの JSON バージョンは、非 JSON バージョンを $util.toJson(...) でラップすることと等価です。例えば、次の 2 つのステートメントはまったく同じです。

```
$util.toStringJson("Hello, World!")
$util.toJson($util.toString("Hello, World!"))
```

toDynamoDB

toDynamoDB utils リスト

$util.dynamodb.toDynamoDB(Object) : Map

入力されたオブジェクトを適切な DynamoDB 表現形式に変換する DynamoDB 用の一般的なオブジェクト変換ツールです。このツールは、一部の型の表現方法に関して融通が利きません。例えば、セット型 ("SS"、"NS"、"BS") ではなくリスト型 ("L") が使用されます。これは、DynamoDB 属性値を記述するオブジェクトを返します。

文字列型の例

```
Input:      $util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

数値型の例

```
Input:      $util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

ブール型の例

```
Input:      $util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

リスト型の例

```
Input:      $util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

マップ型の例

```
Input:      $util.dynamodb.toDynamoDB({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
```

```
                        { "S" : "boop" }
                    ]
                }
            }
        }
```

$util.dynamodb.toDynamoDBJson(Object) : String

$util.dynamodb.toDynamoDB(Object) : Map と同じですが、DynamoDB の属性値を JSON エンコード形式の文字列として返します。

toString utils

toString utils リスト

$util.dynamodb.toString(String) : String

入力文字列を DynamoDB の文字列形式に変換します。このツールは、DynamoDB の属性値を記述したオブジェクトを返します。

```
Input:      $util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

$util.dynamodb.toStringJson(String) : Map

$util.dynamodb.toString(String) : String と同じですが、 DynamoDBの属性値を JSON エンコード形式の文字列として返します。

$util.dynamodb.toStringSet(List<String>) : Map

文字列型のリスト型を DynamoDB の文字列セット形式に変換します。このツールは、DynamoDB の属性値を記述したオブジェクトを返します。

```
Input:      $util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

$util.dynamodb.toStringSetJson(List<String>) : String

$util.dynamodb.toStringSet(List<String>) : Map と同じですが、 DynamoDBの属性値を JSON エンコード形式の文字列として返します。

toNumber utils

toNumber utils リスト

$util.dynamodb.toNumber(Number) : Map

数値を DynamoDB の数値形式に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      $util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

$util.dynamodb.toNumberJson(Number) : String

$util.dynamodb.toNumber(Number) : Map と同じですが、 DynamoDBの属性値を JSON エンコード形式の文字列として返します。

$util.dynamodb.toNumberSet(List<Number>) : Map

数値のリストを DynamoDB の数値セット形式に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      $util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

$util.dynamodb.toNumberSetJson(List<Number>) : String

$util.dynamodb.toNumberSet(List<Number>) : Map と同じですが、 DynamoDBの属性値を JSON エンコード形式の文字列として返します。

toBinary utils

toBinary utils リスト

$util.dynamodb.toBinary(String) : Map

base64 文字列としてエンコードされたバイナリデータを DynamoDB のバイナリ形式に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      $util.dynamodb.toBinary("foo")
```

```
Output:     { "B" : "foo" }
```

$util.dynamodb.toBinaryJson(String) : String

$util.dynamodb.toBinary(String) : Map と同じですが、 DynamoDBの属性値を JSON エンコード形式の文字列として返します。

$util.dynamodb.toBinarySet(List<String>) : Map

base64 文字列としてエンコードされたバイナリデータのリストを DynamoDB のバイナリセット形式に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      $util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

$util.dynamodb.toBinarySetJson(List<String>) : String

$util.dynamodb.toBinarySet(List<String>) : Map と同じですが、 DynamoDBの属性値を JSON エンコード形式の文字列として返します。

toBoolean utils

toBoolean utils リスト

$util.dynamodb.toBoolean(Boolean) : Map

ブール値を DynamoDB の該当するブール形式に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      $util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

$util.dynamodb.toBooleanJson(Boolean) : String

$util.dynamodb.toBoolean(Boolean) : Map と同じですが、 DynamoDBの属性値を JSON エンコード形式の文字列として返します。

toNull utils

toNull utils リスト

`$util.dynamodb.toNull() : Map`

null を DynamoDB の null 形式で返します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      $util.dynamodb.toNull()
Output:     { "NULL" : null }
```

`$util.dynamodb.toNullJson() : String`

`$util.dynamodb.toNull() : Map` と同じですが、 DynamoDBの属性値を JSON エンコード形式の文字列として返します。

toList utils

toList utils リスト

`$util.dynamodb.toList(List) : Map`

オブジェクトのリストを DynamoDB のリスト形式に変換します。リスト内の各項目も、該当する DynamoDB 形式に変換されます。このツールは、一部のネステッドオブジェクトの表現方法に関して融通が利きません。例えば、セット型 ("SS"、"NS"、"BS") ではなくリスト型 ("L") が使用されます。このツールは、DynamoDB の属性値を記述したオブジェクトを返します。

```
Input:      $util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

$util.dynamodb.toListJson(List) : String

$util.dynamodb.toList(List) : Map と同じですが、 DynamoDBの属性値を JSON エンコード形式の文字列として返します。

toMap utils

toMap utils リスト

$util.dynamodb.toMap(Map) : Map

マップを DynamoDB のマップ形式に変換します。マップ内の各値も、該当する DynamoDB 形式に変換されます。このツールは、一部のネステッドオブジェクトの表現方法に関して融通が利きません。例えば、セット型 ("SS"、"NS"、"BS") ではなくリスト型 ("L") が使用されます。このツールは、DynamoDB の属性値を記述したオブジェクトを返します。

```
Input:      $util.dynamodb.toMap({ "foo": "bar", "baz" : 1234, "beep": [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
                           { "S" : "boop" }
                       ]
                   }
               }
           }
```

$util.dynamodb.toMapJson(Map) : String

$util.dynamodb.toMap(Map) : Map と同じですが、 DynamoDBの属性値を JSON エンコード形式の文字列として返します。

$util.dynamodb.toMapValues(Map) : Map

マップ内の各値を該当する DynamoDB 形式に変換して、マップのコピーを作成します。このツールは、一部のネステッドオブジェクトの表現方法に関して融通が利きません。例えば、セット型 ("SS"、"NS"、"BS") ではなくリスト型 ("L") が使用されます。

```
Input:      $util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
```

```
Output:     {
               "foo"  : { "S" : "bar" },
               "baz"  : { "N" : 1234 },
               "beep" : {
                   "L" : [
                       { "S" : "boop" }
                   ]
               }
           }
```

> **Note**
>
> 注 : このツールは `$util.dynamodb.toMap(Map) : Map` と少し異なっていて、属性値全体ではなく DynamoDB の属性値の内容のみを返します。例えば、次の 2 つのステートメントはまったく同じです。
>
> ```
> $util.dynamodb.toMapValues($map)
> $util.dynamodb.toMap($map).get("M")
> ```

`$util.dynamodb.toMapValuesJson(Map) : String`

`$util.dynamodb.toMapValues(Map) : Map` と同じですが、 DynamoDBの属性値を JSON エンコード形式の文字列として返します。

S3Object utils

S3Object utils リスト

`$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`

キー、バケット、およびリージョンを DynamoDB の S3 オブジェクト表現に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      $util.dynamodb.toS3Object("foo", "bar", region = "baz")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo", \"bucket\" : \"bar", \"region
\" : \"baz" } }" }
```

$util.dynamodb.toS3ObjectJson(String key, String bucket, String region) : String

$util.dynamodb.toS3Object(String key, String bucket, String region) : Map と同じですが、 DynamoDBの属性値を JSON エンコード形式の文字列として返します。

$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map

キー、バケット、リージョン、およびバージョン (省略可) を DynamoDB の S3 オブジェクト表現に変換します。これは、DynamoDB 属性値を記述するオブジェクトを返します。

```
Input:      $util.dynamodb.toS3Object("foo", "bar", "baz", "beep")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region
\" : \"baz\", \"version\" = \"beep\" } }" }
```

$util.dynamodb.toS3ObjectJson(String key, String bucket, String region, String version) : String

$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map と同じですが、 DynamoDBの属性値を JSON エンコード形式の文字列として返します。

$util.dynamodb.fromS3ObjectJson(String) : Map

DynamoDB の S3 オブジェクトの文字列値を受け入れて、キー、バケット、リージョン、およびバージョン (省略可) が含まれているマップを返します。

```
Input:      $util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
 \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :
 "beep" }
```

$util.rds の Amazon RDS ヘルパー

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

`$util.rds` には、結果出力から不要なデータを削除して Amazon RDS オペレーションをフォーマットするヘルパーメソッドが含まれています。

$util.rds utils リスト

`$util.rds.toJsonString(String serializedSQLResult): String`

文字列化された生の Amazon Relational Database Service (Amazon RDS) Data API オペレーションの結果形式をより簡潔な文字列に変換することにより `String` を返します。返される文字列は、結果セットの SQL レコードのリストのシリアル化されたリストです。すべてのレコードはキーと値のペアの集合として表されます。キーは対応する列名です。

入力内の対応するステートメントがミューテーションの実行元の SQL クエリ (INSERT、UPDATE、DELETE など) であった場合は、空のリストが返されます。例えば、クエリ `select * from Books limit 2` では、Amazon RDS Data オペレーション からの生の結果が返されます。

```
{
    "sqlStatementResults": [
        {
            "numberOfRecordsUpdated": 0,
            "records": [
                [
                    {
                        "stringValue": "Mark Twain"
                    },
                    {
                        "stringValue": "Adventures of Huckleberry Finn"
                    },
                    {
                        "stringValue": "978-1948132817"
                    }
                ],
                [
                    {
                        "stringValue": "Jack London"
                    },
                    {
                        "stringValue": "The Call of the Wild"
                    },
                    {
                        "stringValue": "978-1948132275"
```

```
                }
              ]
        ],
        "columnMetadata": [
            {
                "isSigned": false,
                "isCurrency": false,
                "label": "author",
                "precision": 200,
                "typeName": "VARCHAR",
                "scale": 0,
                "isAutoIncrement": false,
                "isCaseSensitive": false,
                "schemaName": "",
                "tableName": "Books",
                "type": 12,
                "nullable": 0,
                "arrayBaseColumnType": 0,
                "name": "author"
            },
            {
                "isSigned": false,
                "isCurrency": false,
                "label": "title",
                "precision": 200,
                "typeName": "VARCHAR",
                "scale": 0,
                "isAutoIncrement": false,
                "isCaseSensitive": false,
                "schemaName": "",
                "tableName": "Books",
                "type": 12,
                "nullable": 0,
                "arrayBaseColumnType": 0,
                "name": "title"
            },
            {
                "isSigned": false,
                "isCurrency": false,
                "label": "ISBN-13",
                "precision": 15,
                "typeName": "VARCHAR",
                "scale": 0,
                "isAutoIncrement": false,
```

```
                    "isCaseSensitive": false,
                    "schemaName": "",
                    "tableName": "Books",
                    "type": 12,
                    "nullable": 0,
                    "arrayBaseColumnType": 0,
                    "name": "ISBN-13"
                }
            ]
        }
    ]
}
```

`util.rds.toJsonString` は以下のとおりです。

```
[
  {
    "author": "Mark Twain",
    "title": "Adventures of Huckleberry Finn",
    "ISBN-13": "978-1948132817"
  },
  {
    "author": "Jack London",
    "title": "The Call of the Wild",
    "ISBN-13": "978-1948132275"
  },
]
```

`$util.rds.toJsonObject(String serializedSQLResult): Object`

これは `util.rds.toJsonString` と同じですが、結果は JSON `Object` になります。

$util.http の HTTP ヘルパー

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちらにある APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

$util.http ユーティリティには、HTTP リクエストパラメータの管理やレスポンスヘッダーの追加に使用できるヘルパーメソッドが用意されています。

$util.http utils リスト

$util.http.copyHeaders(Map) : Map

以下の制限された HTTP ヘッダーを除き、マップからヘッダーをコピーします。

- transfer-encoding
- connection
- host
- expect
- keep-alive
- upgrade
- proxy-authenticate
- proxy-authorization
- te
- content-length

このユーティリティを使用して、リクエストヘッダーをダウンストリーム HTTP エンドポイントに転送できます。

```
{
    ...
    "params": {
        ...
        "headers": $util.http.copyHeaders($ctx.request.headers),
        ...
    },
    ...
}
```

$util.http.addResponseHeader(文字列、オブジェクト)

レスポンスの名前 (String) と値 (Object) を含むカスタムヘッダーを 1 つ追加します。以下の制限が適用されます。

- の制限付きヘッダーのリストに加えてcopyHeaders(Map)、ヘッダー名は次のいずれかと一致することはできません。

- Access-Control-Allow-Credentials
- Access-Control-Allow-Origin
- Access-Control-Expose-Headers
- Access-Control-Max-Age
- Access-Control-Allow-Methods
- Access-Control-Allow-Headers
- Vary
- Content-Type

- ヘッダー名は、制限付きプレフィックス x-amzn-または で始めることはできませんx-amz-。
- カスタムレスポンスヘッダーのサイズは 4 KB を超えることはできません。これにはヘッダー名と値が含まれます。
- 各レスポンスヘッダーは、GraphQL 操作ごとに 1 回定義する必要があります。ただし、同じ名前のカスタムヘッダーを複数回定義すると、最新の定義がレスポンスに表示されます。名前に関係なく、すべてのヘッダーがヘッダーサイズの上限に含まれます。
- 名前が空または制限されているヘッダー(String)や null のヘッダー(Object)は無視され、オペレーションのerrors出力に追加されるResponseHeaderErrorエラーが発生します。

```
export function request(ctx) {
  util.http.addResponseHeader('itemsCount', 7)
  util.http.addResponseHeader('render', ctx.args.render)
  return {}
}
```

$util.http.addResponseHeaders(Map)

指定された名前(String)と値のマップ からのレスポンスに複数のレスポンスヘッダーを追加します(Object)。addResponseHeader(String, Object) メソッドにリストされているのと同じ制限が、このメソッドにも適用されます。

```
export function request(ctx) {
  const headers = {
    headerInt: 12,
    headerString: 'stringValue',
    headerObject: {
      field1: 7,
      field2: 'string'
    }
```

```
  }
  util.http.addResponseHeaders(headers)
  return {}
}
```

$util.xml の XML ヘルパー

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちらにある APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

`$util.xml` には、XML レスポンスを JSON またはディクショナリに変換しやすくするヘルパーメソッドが含まれています。

$util.xml utils リスト

`$util.xml.toMap(String) : Map`

XML 文字列を辞書に変換します。

```
Input:

<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
</posts>

Output (JSON representation):

{
  "posts":{
    "post":{
      "id":1,
      "title":"Getting started with GraphQL"
    }
  }
```

```
}


Input:

<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
<post>
  <id>2</id>
  <title>Getting started with AWS AppSync</title>
</post>
</posts>

Output (JSON representation):

{
  "posts":{
    "post":[
        {
          "id":1,
          "title":"Getting started with GraphQL"
        },
        {
          "id":2,
          "title":"Getting started with AWS AppSync"
        }
    ]
  }
}
```

$util.xml.toJsonString(String) : String

XML 文字列を JSON 文字列に変換します。これは、出力が文字列である点を除き、toMap に似ています。これは、XML レスポンスを HTTP オブジェクトから JSON に直接変換し、返す場合に便利です。

$util.xml.toJsonString(String, Boolean) : String

XML 文字列を JSON 文字列に変換します。オプションのブールパラメータを使用して、JSON を文字列エンコードするか判断します。

util.transform の変換ヘルパー

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちらにある APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

`$util.transform` には、Amazon DynamoDB フィルター処理などの、データソースに対する複雑なオペレーションの実行を容易にするヘルパーメソッドが含まれています。

変換ヘルパー

変換ヘルパー utils リスト

`$util.transform.toDynamoDBFilterExpression(Map) : Map`

DynamoDB で使用するために、入力文字列をフィルター式に変換します。

```
Input:

$util.transform.toDynamoDBFilterExpression({
    "title":{
      "contains":"Hello World"
    }
  })

Output:

{
    "expression" : "contains(#title, :title_contains)"
    "expressionNames" : {
        "#title" : "title",
    },
    "expressionValues" : {
        ":title_contains" : { "S" : "Hello World" }
    },
}
```

`$util.transform.toElasticsearchQueryDSL(Map) : Map`

指定された入力を同等の OpenSearch Query DSL 式に変換し、JSON 文字列として返します。

```
Input:

$util.transform.toElasticsearchQueryDSL({
    "upvotes":{
        "ne":15,
        "range":[
            10,
            20
        ]
    },
    "title":{
        "eq":"hihihi",
        "wildcard":"h*i"
    }
  })

Output:
{
    "bool":{
      "must":[
          {
            "bool":{
              "must":[
                  {
                    "bool":{
                      "must_not":{
                        "term":{
                          "upvotes":15
                        }
                      }
                    }
                  },
                  {
                    "range":{
                      "upvotes":{
                        "gte":10,
                        "lte":20
                      }
                    }
                  }
              ]
            }
          },
```

```
          {
            "bool":{
              "must":[
                  {
                    "term":{
                      "title":"hihihi"
                    }
                  },
                  {
                  "wildcard":{
                      "title":"h*i"
                    }
                  }
              ]
            }
          }
      ]
    }
}
```

デフォルトの演算子は AND であると仮定されます。

変換ヘルパーサブスクリプションフィルター

変換ヘルパー、サブスクリプションフィルター、ユーティリティリスト

$util.transform.toSubscriptionFilter(Map) : Map

Map 入力オブジェクトを SubscriptionFilter 式オブジェクトに変換します。$util.transform.toSubscriptionFilter メソッドは$extensions.setSubscriptionFilter() 拡張子への入力として使用されます。詳細については、「拡張子」を参照してください。

$util.transform.toSubscriptionFilter(Map, List) : Map

Map 入力オブジェクトを SubscriptionFilter 式オブジェクトに変換します。$util.transform.toSubscriptionFilter メソッドは$extensions.setSubscriptionFilter() 拡張子への入力として使用されます。詳細については、「拡張子の使用」を参照してください。

1 番目の引数は、SubscriptionFilter 式オブジェクトに変換される Map 入力オブジェクトです。2 番目の引数は、SubscriptionFilter 式オブジェクトを作成する際に 1 番目の Map 入力オブジェクトでは無視されるフィールド名の List です。

$util.transform.toSubscriptionFilter(Map, List, Map) : Map

Map 入力オブジェクトを SubscriptionFilter 式オブジェクトに変換します。$util.transform.toSubscriptionFilter メソッドは$extensions.setSubscriptionFilter() 拡張子への入力として使用されます。詳細については、「拡張子」を参照してください。

1 番目の引数は SubscriptionFilter 式オブジェクトに変換される Map 入力オブジェクト、2 番目の引数は最初の Map 入力オブジェクトでは無視されるフィールド名のList、3 番目の引数は SubscriptionFilter 式オブジェクトの作成時に含まれる厳密な規則の Map 入力オブジェクトです。これらの厳密なルールは、少なくとも 1 つのルールが満たされてサブスクリプションフィルターを通過するように SubscriptionFilter 式オブジェクトに含まれています。

サブスクリプションフィルター引数

以下の表では、以下のユーティリティの引数の定義方法について説明しています。

- $util.transform.toSubscriptionFilter(Map) : Map
- $util.transform.toSubscriptionFilter(Map, List) : Map
- $util.transform.toSubscriptionFilter(Map, List, Map) : Map

Argument 1: Map

引数 1 は、以下のキー値を持つ Map オブジェクトです。

- フィールド名
- "and"
- "or"

フィールド名をキーとする場合、これらのフィールドのエントリの条件は "operator" : "value" という形式になります。

次の例では、Map にエントリを追加する方法を示します。

```
"field_name" : {
                    "operator1" : value
               }

## We can have multiple conditions for the same field_name:

"field_name" : {
                    "operator1" : value
                    "operator2" : value
                    .
                    .
                    .
               }
```

フィールドに 2 つ以上の条件が設定されている場合、これらの条件はすべて OR 演算を使用するものとみなされます。

入力 Map には「and」と「or」をキーとして使用することもできます。つまり、その中のすべてのエントリは、キーに応じて AND または OR ロジックを使用して結合する必要があります。キー値「and」と「or」には条件の配列が必要です。

```
"and" : [

            {
                "field_name1" : {
                    "operator1" : value
                }
             },

             {
                "field_name2" : {
                    "operator1" : value
                }
             },
             .
             .
        ].
```

「and」と「or」はネストできることに注意してください。つまり、「and/or」を別の「and/or」ブロック内にネストしてもかまいません。ただし、これは単純なフィールドでは機能しません。

```
"and" : [
```

```
            {
                "field_name1" : {
                    "operator" : value
                }
             },

             {
                "or" : [
                            {
                                "field_name2" : {
                                    "operator" : value
                                }
                            },

                            {
                                "field_name3" : {
                                    "operator" : value
                                }
                            }

                        ].
```

次の例は、$util.transform.toSubscriptionFilter(Map) : Mapを使用して引数 1 を入力したものです。

入力

引数 1: マップ:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
```

```
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

出力

結果は Map オブジェクトです。

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "author",
          "operator": "eq",
```

```
          "value": "Admin"
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "gte",
          "value": 20
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
```

```
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Admin"
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "gte",
          "value": 20
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 2: List

引数 2 には、SubscriptionFilter 式オブジェクトを作成するときに入力 Map (引数 1) で考慮してはいけないフィールド名の List が含まれています。List は空になることもあります。

次の例は、$util.transform.toSubscriptionFilter(Map, List) : Map を使用した引数 1 と引数 2 の入力を示しています。

入力

引数 1: マップ:

```
{

  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

引数 2: リスト:

```
["percentageUp", "author"]
```

出力

結果は Map オブジェクトです。

```
{
```

```
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 3: Map

引数 3 は、フィールド名をキー値とする Map オブジェクトです (「and」や「or」は使用できません)。フィールド名がキーの場合、これらのフィールドの条件は "operator" : "value" という形式のエントリになります。引数 1 とは異なり、引数 3 では同じキーに複数の条件を設定できません。さらに、引数 3 には「and」や「or」句がないため、ネストも必要ありません。

引数 3 は厳密な規則のリストを表し、これらの条件の少なくとも 1 つが満たされてフィルタを通過するように SubscriptionFilter 式オブジェクトに追加されます。

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
```

```
.
```

次の例は、$util.transform.toSubscriptionFilter(Map, List, Map) : Map を使用した引数 1、引数 2、引数 3 の入力を示しています。

入力

引数 1: マップ:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

引数 2: リスト:

```
["percentageUp", "author"]
```

引数 3: マップ:

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
}
```

出力

結果は Map オブジェクトです。

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        },
        {
          "fieldName": "upvotes",
          "operator": "gte",
          "value": 250
        }
      ]
```

```
    },
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Person1"
        }
      ]
    }
  ]
}
```

$util.math の math ヘルパー

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちらにある APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

`$util.math` には一般的な算術演算を支援するメソッドが含まれています。

$util.math utils リスト

`$util.math.roundNum(Double) : Integer`

倍精度値 をとり、最も近い整数に四捨五入します。

`$util.math.minVal(Double, Double) : Double`

2 つの倍精度値を取り、2 つの倍精度浮動小数点間の最小値を返します。

`$util.math.maxVal(Double, Double) : Double`

2 つの倍精度値を取り、2 つの倍精度浮動小数点間の最大値を返します。

`$util.math.randomDouble() : Double`

0 から 1 までの乱数を返します。

> **Important**
>
> この関数は、エントロピーランダム性が高い場合 (暗号など) には使用しないでください。

`$util.math.randomWithinRange(Integer, Integer) : Integer`

指定された範囲内のランダムな整数値を返します。最初の引数は範囲の下限値を指定し、2 番目の引数は範囲の上限値を指定します。

> **Important**
>
> この関数は、エントロピーランダム性が高い場合 (暗号など) には使用しないでください。

$util.str 内の文字列ヘルパー

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

`$util.str` には一般的な文字列操作を支援するメソッドが含まれています。

$util.str utils リスト

`$util.str.toUpper(String) : String`

文字列を受け取り、それをすべて大文字に変換します。

`$util.str.toLower(String) : String`

文字列を受け取り、それをすべて小文字に変換します。

`$util.str.toReplace(String, String, String) : String`

文字列内の部分文字列を別の文字列に置き換えます。最初の引数は、置換操作を実行する文字列を指定します。2 番目の引数は、置換する部分文字列を指定します。3 番目の引数は、2 番目の引数を置き換える文字列を指定します。以下は、このユーティリティの使用例です。

```
INPUT:      $util.str.toReplace("hello world", "hello", "mellow")
OUTPUT:     "mellow world"
```

`$util.str.normalize(String, String) : String`

NFC、NFD、NFKC、または NFKD の 4 つのユニコード正規化形式のいずれかを使用して文字列を正規化します。最初の引数は正規化する文字列です。2 番目の引数は、正規化プロセスに使用する正規化タイプを指定する「nfc」「nfd」「nfkc」「nfkd」のいずれかです。

拡張子

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

`$extensions` には、リゾルバー内で追加のアクションを行うためのメソッドセットが含まれています。

拡張子の使用

`$extensions.evictFromApiCache(String, String, Object) : Object`

AWS AppSync サーバー側のキャッシュから項目を削除します。最初の引数は型名です。2 番目の引数はフィールド名です。3 番目の引数は、キャッシュキー値を指定するキーと値のペア項目を含むオブジェクトです。オブジェクト内の項目は、キャッシュされたリゾルバー `cachingKey` のキャッシュキーと同じ順序で配置する必要があります。

> **Note**
>
> このユーティリティはミューテーションに対してのみ機能し、クエリには使用できません。

サブスクリプションの延長

`$extensions.setSubscriptionFilter(filterJsonObject)`

強化されたサブスクリプションフィルターを定義します。各サブスクリプション通知イベントは、提供されたサブスクリプションフィルタに対して評価され、すべてのフィルタが `true` に評価された場合、クライアントに通知を配信します。引数は以下のセクションで説明するとおり `filterJsonObject` です。

> **Note**
>
> この拡張メソッドは、サブスクリプションリゾルバーのレスポンスマッピングテンプレートでのみ使用できます。

`$extensions.setSubscriptionInvalidationFilter(filterJsonObject)`

サブスクリプション無効化フィルターを定義します。サブスクリプションフィルタは無効化ペイロードに照らして評価されてから、フィルターが `true` と評価された場合、与えられたサブスクリプションを無効にします。引数は以下のセクションで説明するとおり `filterJsonObject` です。

> **Note**
>
> この拡張メソッドは、サブスクリプションリゾルバーのレスポンスマッピングテンプレートでのみ使用できます。

`$extensions.invalidateSubscriptions(invalidationJsonObject)`

ミューテーションによるサブスクリプションの無効化を開始するために使用されます。引数は以下のセクションで説明するとおり `invalidationJsonObject` です。

> **Note**
>
> この拡張関数はミューテーションリゾルバーのレスポンスマッピングテンプレートでのみ使用できます。
> 1 つのリクエストで使用できるユニークな `$extensions.invalidateSubscriptions()` メソッド呼び出しは 5 つまでです。この制限を超えた場合、GraphQL エラーが表示されます。

引数: filterJsonObject

JSON オブジェクトは、サブスクリプションフィルターまたは無効化フィルターのいずれかを定義します。これは、`filterGroup` 内のフィルターの配列です。各フィルターは個別のフィルターの集まりです。

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
```

```
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
            ]

        }
    ]
}
```

各フィルターには次の 3 つの属性があります。

- `fieldName` – GraphQL スキーマフィールド。
- `operator` – オペレータータイプ。
- `value` – サブスクリプション通知 `fieldName` 値と比較する値。

以下は、これらの属性の割り当ての例です。

```
{
 "fieldName" : "severity",
 "operator" : "le",
 "value" : $context.result.severity
}
```

フィールド: fieldName

文字列タイプ `fieldName` は、サブスクリプション通知ペイロード内の `fieldName` と一致する GraphQL スキーマで定義されているフィールドを指します。一致するものが見つかると、GraphQL スキーマフィールドの `value` がサブスクリプション通知フィルターのフィールドの `value` と比較されます。次の例では、`fieldName` フィルターは特定の GraphQL タイプで定義された `service` フィールドと一致します。通知ペイロードが AWS AppSync と等しい `value` のある `service` フィールドを含む場合、フィルタは `true` と評価されます :

```
{
 "fieldName" : "service",
 "operator" : "eq",
 "value" : "AWS AppSync"
}
```

フィールド: 値

値は演算子によって異なるタイプでもかまいません。

- 1 つの数値またはブール値
  - 文字列型の例: "test"、"service"
  - 数値型の例:
  - ブール型の例:
- 数字または文字列のペア
  - 文字列ペアの例: ["test1","test2"]、["start","end"]
  - 数値ペアの例: [1,4]、[67,89]、[12.45, 95.45]
- 数値または文字列の配列
  - 文字列配列の例:["test1","test2","test3","test4","test5"]
  - 数値配列の例: [1,2,3,4,5]、[12.11,46.13,45.09,12.54,13.89]

フィールド演算子

大文字と小文字が区別される文字列で、以下の値を指定できます。

演算子	説明	可能な値型
eq	Equal	integer, float, string, Boolean
ne	Not equal	integer, float, string, Boolean
le	Less than or equal	integer, float, string
lt	Less than	integer, float, string
ge	Greater than or equal	integer, float, string
gt	Greater than	integer, float, string
contains	Checks for a subsequence or value in the set.	integer, float, string

notContains	Checks for the absence of a subsequence or absence of a value in the set.	integer, float, string
beginsWith	Checks for a prefix.	string
in	Checks for matching elements that are in the list.	Array of integer, float, or string
notIn	Checks for matching elements that aren't in the list.	Array of integer, float, or string
between	Between two values	integer, float, string
containsAny	Contains common elements	integer, float, string

次の表は、サブスクリプション通知での各オペレーターの使用法を示したものです。

eq (equal)

eq 演算子は、サブスクリプション通知フィールドの値がフィルターの値と一致し、厳密に等しいかどうかを true に対して評価します。次の例では、フィルターはサブスクリプション通知に AWS AppSync に等しい値の service フィールドがあるかどうかを true に対して評価します。

指定できる値のタイプ: 整数、浮動小数点、文字列、ブール値

```
{
 "fieldName" : "service",
 "operator" : "eq",
 "value" : "AWS AppSync"
}
```

ne (not equal)

ne 演算子は、サブスクリプション通知フィールドの値がフィルターの値と異なるかどうかを true に対して評価します。次の例では、フィルターは、サブスクリプション通知に AWS AppSync と異なる値の service フィールドがあるかどうかを true に対して評価します。

指定できる値のタイプ: 整数、浮動小数点、文字列、ブール値

```
{
 "fieldName" : "service",
 "operator" : "ne",
 "value" : "AWS AppSync"
}
```

le (less or equal)

le 演算子は、サブスクリプション通知フィールドの値がフィルターの値以下かどうかを true に対して評価します。次の例では、フィルターは、サブスクリプション通知に5以下の値のsizeフィールドがあるかどうかをtrue に対して評価します。

指定できる値のタイプ: 整数、浮動小数点、文字列

```
{
 "fieldName" : "size",
 "operator" : "le",
 "value" : 5
}
```

lt (less than)

lt 演算子は、サブスクリプション通知フィールドの値がフィルターの値よりも小さいかどうかを true に対して評価します。次の例では、フィルターは、サブスクリプション通知に 5 以下の値の size フィールドがあるかどうかを true に対して評価します。

指定できる値のタイプ: 整数、浮動小数点、文字列

```
{
 "fieldName" : "size",
 "operator" : "lt",
 "value" : 5
}
```

ge (greater or equal)

ge 演算子は、true サブスクリプション通知フィールドの値がフィルターの値以上かどうかを評価します。次の例では、フィルターは、サブスクリプション通知に 5 以上の値の size フィールドがあるかどうかを true に対して評価します。

指定できる値のタイプ: 整数、浮動小数点、文字列

```
{
 "fieldName" : "size",
 "operator" : "ge",
 "value" : 5
}
```

gt (greater than)

gt 演算子は、サブスクリプション通知フィールドの値がフィルターの値より大きいかどうかを に対して評価します。次の例では、フィルターは、サブスクリプション通知に 5 以上の値の size フィールドがあるかどうかを true に対して評価します。

指定できる値のタイプ: 整数、浮動小数点、文字列

```
{
 "fieldName" : "size",
 "operator" : "gt",
 "value" : 5
}
```

contains

contains 演算子は、サブストリング、サブシーケンス、またはセットまたは単一アイテムをチェックします。contains 演算子を含むフィルターは、サブスクリプション通知フィールドの値にフィルタ値が含まれているかどうかを true に対して評価します。次の例では、フィルターは、サブスクリプション通知に値 10 を含む配列値を持つ seats フィールドがあるかどうかを true に対して評価します。

指定できる値のタイプ: 整数、浮動小数点、文字列

```
{
 "fieldName" : "seats",
 "operator" : "contains",
 "value" : 10
}
```

別の例として、フィルターはサブスクリプション通知にサブストリングとして launch を含む event フィールドがあるかどうかをtrue に対して評価します。

```
{
 "fieldName" : "event",
```

```
 "operator" : "contains",
 "value" : "launch"
}
```

notContains

notContains 演算子は、サブストリング、サブシーケンス、またはセットまたは値がないことをチェックします。notContains 演算子を使ったフィルターは、サブスクリプション通知フィールドの値にフィルタ値が含まれていないかどうかを true に対して評価します。次の例では、フィルターは、サブスクリプション通知に値 10 を含まない配列値の seats フィールドがあるかどうかを true に対して評価します。

指定できる値のタイプ: 整数、浮動小数点、文字列

```
{
 "fieldName" : "seats",
 "operator" : "notContains",
 "value" : 10
}
```

別の例として、フィルターはサブスクリプション通知のサブシーケンスとしての launch なしで event フィールド値があるかどうかを true に対して評価します。

```
{
 "fieldName" : "event",
 "operator" : "notContains",
 "value" : "launch"
}
```

beginsWith

beginsWith 演算子は文字列のプレフィックスをチェックします。beginsWith 演算子を含むフィルターは、サブスクリプション通知フィールドの値がフィルタの値で始まるかどうかを true に対して評価します。次の例では、フィルターは、サブスクリプション通知に値がで始まる service フィールドがあるかどうかを true に対して評価します。

指定できる値の種類: 文字列

```
{
 "fieldName" : "service",
 "operator" : "beginsWith",
 "value" : "AWS"
```

```
}
```

in

in 演算子は配列内の一致する要素をチェックします。in 演算子を含むフィルタは、サブスクリプション通知フィールド値が配列に存在するかどうかを true に対して評価します。次の例では、フィルターは、サブスクリプション通知に配列 [1,2,3] 内のいずれかの値を含む severity フィールドがあるかどうかを true に対して評価します。

指定できる値のタイプ: 整数、浮動小数点数、または文字列の配列

```
{
 "fieldName" : "severity",
 "operator" : "in",
 "value" : [1,2,3]
}
```

notIn

notIn 演算子は配列に欠けている要素がないかチェックします。notIn 演算子を含むフィルタは、サブスクリプション通知フィールド値が配列に存在しないかどうかを true に対して評価します。次の例では、フィルターは、サブスクリプション通知に、配列 [1,2,3] に存在しない値のいずれかを含む severity フィールドがあるかどうかを true に対して評価します。

指定できる値のタイプ: 整数、浮動小数点数、または文字列の配列

```
{
 "fieldName" : "severity",
 "operator" : "notIn",
 "value" : [1,2,3]
}
```

between

between 演算子は 2 つの数値または文字列の間の値をチェックします。between 演算子を含むフィルターは、サブスクリプション通知フィールドの値がフィルタの値ペアの間にあるかどうかを true に対して評価します。次の例では、フィルターは、サブスクリプション通知に値 2、3、4 を含む severity フィールドがあるかどうかを true に対して評価します。

指定できる値のタイプ: 整数、浮動小数点数、または文字列のペア

```
{
```

```
 "fieldName" : "severity",
 "operator" : "between",
 "value" : [1,5]
}
```

containsAny

containsAny 演算子は配列内の共通要素をチェックします。containsAny 演算子を使ったフィルタ = は、サブスクリプション通知フィールドの設定値とフィルタセット値の共通部分が空でないかどうかを true に対して評価します。次の例では、フィルターは、サブスクリプション通知にまたはを含む配列値を持つ seats フィールドがあるかどうかをtrueに対して評価します。つまり、フィルターは、サブスクリプション通知の [15,20,30] フィールド値が true または seats であるかどうかを [10,11] に対して評価します。

指定できる値のタイプ: 整数、浮動小数点数、または文字列

```
{
 "fieldName" : "seats",
 "operator" : "containsAny",
 "value" : [10, 15]
}
```

AND ロジック

filterGroup 配列内の filters オブジェクト内に複数のエントリを定義することで、AND ロジックを使用して複数のフィルターを組み合わせることができます。次の例では、フィルターは、サブスクリプション通知にフィールドの値が等しく、userId フィールド値が 1 または group のいずれかであるかどうかを true に対して評価します。

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "in",
```

```
                    "value" : ["Admin", "Developer"]
                }
            ]

        }
    ]
}
```

OR ロジック

filterGroup 配列内に複数のフィルターオブジェクトを定義することで、OR ロジックを使用して複数のフィルターを組み合わせることができます。次の例では、フィルターは、サブスクリプション通知に Admin OR Developer と同等の値の userId フィールドがあるかどうか、1 またはの group フィールド値があるかどうかを true に対して評価します。

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

例外

フィルターの使用にはいくつかの制限があることに注意してください。

- filters オブジェクトには、フィルターごとに最大 5 つのユニークな fieldName 項目を設定できます。つまり、AND ロジックを使用して最大 5 つの個別の fieldName オブジェクトを組み合わせることができます。
- containsAny 演算子には最大 20 個の値を指定できます。
- in and notIn 演算子には最大 5 つの値を指定できます。
- 接続文字列は最大 255 文字です。
- 文字列比較では、大文字と小文字を区別します。
- ネストされたオブジェクトフィルタリングでは、最大 5 つのネストレベルのフィルタリングが可能です。
- filterGroup にはそれぞれ、最大 10 個 filters 設定できます。つまり、OR ロジックを使用して最大 10 個 filters を組み合わせることができます。
  - in 演算子は OR ロジックの特殊なケースです。次の例には、次の 2 つの filters があります。

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]
        }
    ]
}
```

前述のフィルターグループは次のように評価され、最大フィルター制限に加算されます。

```
{
    "filterGroup": [
        {
           "filters" : [
```

```
                {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "eq",
                    "value" : "Admin"
                }
            ]
        },
        {
            "filters" : [
                {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "eq",
                    "value" : "Developer"
                }
            ]
        }
    ]
}
```

引数: invalidationJsonObject

`invalidationJsonObject` では以下が定義されます。

- `subscriptionField` – 無効にする GraphQL スキーマのサブスクリプション。`subscriptionField`で文字列として定義されている 1 つのサブスクリプションは無効とみなされます。
- `payload` — 無効化フィルターがその値に対して `true` と評価された場合に、サブスクリプションを無効化するための入力として使用されるキーと値のペアリスト。

以下の例では、サブスクリプションリゾルバーで定義された無効化フィルターが payload 値に対して true と評価されたとき、onUserDelete サブスクリプションを使用してサブスクライブして接続しているクライアントを無効にします。

```
$extensions.invalidateSubscriptions({
        "subscriptionField": "onUserDelete",
        "payload": {
                "group": "Developer"
                "type" : "Full-Time"
      }
    })
```

DynamoDB の AWS AppSync リゾルバーマッピングテンプレートリファレンス

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync DynamoDB 関数を使用すると、受信 GraphQL GraphQL リクエストを DynamoDB 呼び出しにマッピングし、DynamoDB レスポンスを GraphQL にマッピングすることで、GraphQL を使用してアカウント内の既存の Amazon DynamoDB テーブルにデータを保存および取得できます。このセクションでは、サポートされる DynamoDB オペレーションのリクエストハンドラーとレスポンスハンドラーについて説明します。

- GetItem - GetItem リクエストでは、DynamoDB 関数に DynamoDB への GetItem リクエストを行うように指示し、DynamoDB 内の項目のキーと、一貫した読み取りを使用するかどうかを指定できます。
- PutItem - PutItem リクエストマッピングドキュメントでは、DynamoDB 関数に DynamoDB への PutItem リクエストを行うように指示し、DynamoDB の項目のキー、項目の全内容 (キーと attributeValues で構成される)、およびオペレーションが成功するための条件を指定できます。
- UpdateItem - UpdateItem リクエストでは、DynamoDB 関数に DynamoDB への UpdateItem リクエストを行うように指示し、DynamoDB の項目のキー、DynamoDB の項目を更新する方法を説明する更新式、およびオペレーションが成功するための条件を指定できます。

- DeleteItem - DeleteItem リクエストでは、DynamoDB 関数に DynamoDB に DeleteItem リクエストを行うように指示し、DynamoDB 内の項目のキーと、オペレーションが成功するための条件を指定できます。
- Query - Query リクエストオブジェクトを使用すると、DynamoDB リゾルバーに DynamoDB へのクエリリクエストを行うように指示できます。また、キー式、使用するインデックス、追加のフィルター、返す項目の数、一貫した読み取りを使用するかどうか、クエリの方向 (前方または後方)、ページ分割トークンを指定できます。
- Scan - Scan リクエストは、DynamoDB 関数に DynamoDB への Scan リクエストを指示することができ、結果を除外するフィルタ、使用するインデックス、返すアイテムの数、一貫性のある読み取りを使用するかどうか、ページ分割トークン、並列スキャンを指定することができます。
- Sync - Sync リクエストオブジェクトを使用すると、DynamoDB テーブルからすべての結果を取得し、最後のクエリ (差分更新) 以降に変更されたデータのみを受け取ることができます。Sync リクエストは、バージョン管理された DynamoDB データソースに対してのみ行うことができます。フィルタを指定して、結果、返す項目の数、ページ分割トークン、および最後の Sync オペレーションが開始された日時を除外できます。
- BatchGetItem - BatchGetItem リクエストオブジェクトを使用すると、DynamoDB の関数から DynamoDB への BatchGetItem リクエストで、複数のテーブルから複数の項目を取得するように指定できます。このリクエストオブジェクトでは、項目を取得するテーブル名と、各テーブルから取得する項目のキーを指定する必要があります。
- BatchDeleteItem - BatchDeleteItem リクエストオブジェクトを使用すると、DynamoDB の関数から DynamoDB への BatchWriteItem リクエストで、複数のテーブルから複数の項目を削除するように指定できます。このリクエストオブジェクトでは、項目を削除するテーブル名と、各テーブルから削除する項目のキーを指定する必要があります。
- BatchPutItem - BatchPutItem リクエストオブジェクトを使用すると、DynamoDB の関数から DynamoDB への BatchWriteItem リクエストオブジェクトを使用すると リクエストで、複数のテーブルに複数の項目を配置するように指定できます。このリクエストオブジェクトでは、項目を配置するテーブル名と、各テーブルに配置する完全な項目を指定する必要があります。
- TransactGetItems - TransactGetItems リクエストオブジェクトを使用すると、DynamoDB の関数から DynamoDB への TransactGetItems リクエストで、複数のテーブルから複数の項目を取得するように指定できます。このリクエストオブジェクトでは、項目を取得する各リクエスト項目のテーブル名と、各テーブルから取得する各リクエスト項目のキーを指定する必要があります。
- TransactWriteItems - TransactWriteItems リクエストオブジェクトを使用すると、DynamoDB の関数から DynamoDB への TransactWriteItems リクエストで、複数のテーブルに複数の項目を書き込むように指定できます。このリクエストオブジェクトでは、各リクエスト項目の宛先テーブル名、

実行する各リクエスト項目のオペレーション、および書き込む各リクエスト項目のキーを指定する必要があります。

- 型システム (リクエストマッピング) - DynamoDB 型付けが AWS AppSync リクエストに統合される方法の詳細について説明します。
- 型システム (レスポンスマッピング) - DynamoDB タイプがレスポンスペイロードで GraphQL または JSON に自動的に変換される方法について詳しく説明します。
- フィルター - クエリおよびスキャンオペレーションのフィルターの詳細について説明します。
- 条件式 - PutItem オペレーション、UpdateItem オペレーション、DeleteItem オペレーションの条件式について説明します。
- トランザクション条件式 - TransactWriteItems オペレーションの条件式について説明します。
- プロジェクション - 読み取りオペレーションで属性を指定する方法について説明します。

GetItem

`GetItem` リクエストマッピングドキュメントでは、 AWS AppSync DynamoDB リゾルバーに DynamoDB への`GetItem`リクエストを行うように指示し、以下を指定できます。

- DynamoDB の項目のキー
- 整合性のある読み込みを使用するかどうか

`GetItem` マッピングドキュメントの構造は次のとおりです。

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true,
    "projection" : {
        ...
    }
}
```

各フィールドの定義は以下のようになります。

GetItem フィールド

GetItem フィールドリスト

`version`

テンプレート定義バージョン `2017-02-28` と `2018-05-29` は現在サポートされています。この値は必須です。

`operation`

実行する DynamoDB の処理。`GetItem` DynamoDB の処理を実行するには、これを `GetItem` に設定する必要があります。この値は必須です。

`key`

DynamoDB の項目のキー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

`consistentRead`

DynamoDB で強力な整合性のある読み込みを実行するかどうかを示します。これはオプションであり、デフォルトは `false` です。

`projection`

DynamoDB オペレーションから返される属性を指定するために使用されるプロジェクション。プロジェクションの詳細については、「プロジェクション」を参照してください。このフィールドはオプションです。

DynamoDB から返された項目が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、マッピングコンテキスト (`$context.result`) で参照できます。

DynamoDB の型変換の詳細については、「型システム (リクエストマッピング)」を参照してください。

レスポンスマッピングテンプレートの詳細については、「リゾルバーのマッピングテンプレートの概要」を参照してください。

例

以下の例は、GraphQL クエリ `getThing(foo: String!, bar: String!)` のマッピングテンプレートです。

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
        "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
    },
    "consistentRead" : true
}
```

DynamoDB `GetItem` API の詳細については、「DynamoDB API のドキュメント」を参照してください。

PutItem

`PutItem` リクエストマッピングドキュメントでは、 AWS AppSync DynamoDB リゾルバーに DynamoDB への`PutItem`リクエストを行うように指示し、以下を指定できます。

- DynamoDB の項目のキー
- 項目の完全なコンテンツ (`key` および `attributeValues` で構成されます)
- 処理が成功する条件

`PutItem` マッピングドキュメントの構造は次のとおりです。

```
{
    "version" : "2018-05-29",
    "operation" : "PutItem",
    "customPartitionKey" : "foo",
    "populateIndexFields" : boolean value,
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "attributeValues" : {
        "baz" : ... typed value
    },
```

```
    "condition" : {
       ...
    },
    "_version" : 1
}
```

各フィールドの定義は以下のようになります。

PutItem フィールド

PutItem フィールドリスト

`version`

テンプレート定義バージョン `2017-02-28` と `2018-05-29` は現在サポートされています。この値は必須です。

`operation`

実行する DynamoDB の処理。`PutItem` DynamoDB の処理を実行するには、これを `PutItem` に設定する必要があります。この値は必須です。

`key`

DynamoDB の項目のキー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

`attributeValues`

DynamoDB に渡す項目の残りの属性です。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。このフィールドはオプションです。

`condition`

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件が指定されていない場合、`PutItem` リクエストはその項目の既存のエントリを上書きします。条件の詳細については、「条件式」を参照してください。この値はオプションです。

`_version`

項目の既知の最新バージョンを表す数値。この値はオプションです。このフィールドは競合の検出に使用され、バージョン管理されたデータソースでのみサポートされます。

customPartitionKey

有効にすると、この文字列値は、バージョニングが有効になっているときにデルタ同期テーブルで使用される ds_sk および ds_pk レコードの形式を変更します (詳細については、「AWS AppSyncデベロッパーガイド」の「競合検出と同期」を参照)。有効にすると、populateIndexFields エントリの処理も有効になります。このフィールドはオプションです。

populateIndexFields

ブール値で、**customPartitionKey** と一緒に有効にすると、差分同期テーブル、具体的には gsi_ds_pk と gsi_ds_sk 列のレコードごとに新しいエントリが作成されます。詳細については、「AWS AppSyncデベロッパーガイド」の「競合検出と同期」を参照してください。このフィールドはオプションです。

DynamoDB に書き込まれた項目が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、マッピングコンテキスト ($context.result) で参照できます。

DynamoDB の型変換の詳細については、「型システム (リクエストマッピング)」を参照してください。

レスポンスマッピングテンプレートの詳細については、「リゾルバーのマッピングテンプレートの概要」を参照してください。

例 1

以下は、GraphQL ミューテーション updateThing(foo: String!, bar: String!, name: String!, version: Int!) のマッピングテンプレートです。

指定したキーに対応する項目がない場合は、作成されます。指定したキーに対応する項目がすでにある場合は、上書きされます。

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
        "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
    },
    "attributeValues" : {
        "name"    : $util.dynamodb.toDynamoDBJson($ctx.args.name),
```

```
        "version" : $util.dynamodb.toDynamoDBJson($ctx.args.version)
    }
}
```

例 2

以下は、GraphQL ミューテーション `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)` のマッピングテンプレートです。

この例では、DynamoDB に現在ある項目の `version` フィールドに `expectedVersion` が設定されていることを確認します。

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
        "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
    },
    "attributeValues" : {
        "name"    : $util.dynamodb.toDynamoDBJson($ctx.args.name),
        #set( $newVersion = $context.arguments.expectedVersion + 1 )
        "version" : $util.dynamodb.toDynamoDBJson($newVersion)
    },
    "condition" : {
        "expression" : "version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" : $util.dynamodb.toDynamoDBJson($expectedVersion)
        }
    }
}
```

DynamoDB `PutItem` API の詳細については、「DynamoDB API のドキュメント」を参照してください。

UpdateItem

`UpdateItem` リクエストマッピングドキュメントでは、 AWS の AppSync DynamoDB リゾルバーから DynamoDB への `UpdateItem` リクエストを定義し、以下のように指定できます。

- DynamoDB の項目のキー
- DynamoDB の項目を更新する方法を示す更新式

- 処理が成功する条件

UpdateItem マッピングドキュメントの構造は次のとおりです。

```
{
    "version" : "2018-05-29",
    "operation" : "UpdateItem",
    "customPartitionKey" : "foo",
    "populateIndexFields" : boolean value,
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "update" : {
        "expression" : "someExpression",
        "expressionNames" : {
           "#foo" : "foo"
       },
       "expressionValues" : {
           ":bar" : ... typed value
       }
    },
    "condition" : {
        ...
    },
    "_version" : 1
}
```

各フィールドの定義は以下のようになります。

UpdateItem フィールド

UpdateItem フィールドリスト

`version`

テンプレート定義バージョン `2017-02-28` と `2018-05-29` は現在サポートされています。この値は必須です。

`operation`

実行する DynamoDB の処理。`UpdateItem` DynamoDB の処理を実行するには、これを `UpdateItem` に設定する必要があります。この値は必須です。

key

DynamoDB の項目のキー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

update

update セクションには、 DynamoDBの項目の更新方法を示す更新式を指定することができます。更新式の記述方法の詳細については、「DynamoDB UpdateExpressions」のドキュメントを参照してください。このセクションは必須です。

update セクションには次の 3 つのコンポーネントがあります。

expression

更新式です。この値は必須です。

expressionNames

式の属性名のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される名前のプレースホルダーに対応し、値は DynamoDB の項目の属性名と一致する文字列でなければなりません。このフィールドはオプションであり、expression で使用される式の属性名のプレースホルダーのみを入力します。

expressionValues

式の属性値のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される値のプレースホルダーに対応し、値は型付き値でなければなりません。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この指定は必須です。このフィールドはオプションであり、expression で使用される式の属性値のプレースホルダーのみを入力します。

condition

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件を指定していない場合は、UpdateItem リクエストによって、現在の状態にかかわらず、既存のエントリが更新されます。条件の詳細については、「条件式」を参照してください。この値はオプションです。

_version

項目の既知の最新バージョンを表す数値。この値はオプションです。このフィールドは競合の検出に使用され、バージョン管理されたデータソースでのみサポートされます。

customPartitionKey

有効にすると、この文字列値は、バージョニングが有効になっているときにデルタ同期テーブルで使用される ds_sk および ds_pk レコードの形式を変更します (詳細については、「AWS AppSyncデベロッパーガイド」の「競合検出と同期」を参照)。有効にすると、populateIndexFields エントリの処理も有効になります。このフィールドはオプションです。

populateIndexFields

ブール値で、**customPartitionKey** と一緒に有効にすると、差分同期テーブル、具体的には gsi_ds_pk と gsi_ds_sk 列のレコードごとに新しいエントリが作成されます。詳細については、「AWS AppSyncデベロッパーガイド」の「競合検出と同期」を参照してください。このフィールドはオプションです。

DynamoDB の更新された項目が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、マッピングコンテキスト ($context.result) で参照できます。

DynamoDB の型変換の詳細については、「型システム (リクエストマッピング)」を参照してください。

レスポンスマッピングテンプレートの詳細については、「リゾルバーのマッピングテンプレートの概要」を参照してください。

例 1

次の例は、GraphQL ミューテーション upvote(id: ID!) のマッピングテンプレートです。

この例では、DynamoDB の項目の upvotes フィールドと version フィールドが 1 ずつ増加されます。

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "update" : {
        "expression" : "ADD #votefield :plusOne, version :plusOne",
        "expressionNames" : {
            "#votefield" : "upvotes"
```

```
        },
        "expressionValues" : {
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

例 2

以下は、GraphQL ミューテーション `updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)` のマッピングテンプレートです。

これは、引数を確認して、クライアントから入力された引数のみを含む更新式を動的に生成する複雑な例です。たとえば、`title` と `author` を省略すると、それらは更新されません。引数が指定されているが、その値が `null` の場合、そのフィールドは DynamoDB のオブジェクトから削除されます。最後に、この処理内の条件によって、DynamoDB に現在ある項目の `version` フィールドに `expectedVersion` が設定されているかどうかを確認します。

```
{
    "version" : "2017-02-28",

    "operation" : "UpdateItem",

    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },

    ## Set up some space to keep track of things we're updating **
    #set( $expNames  = {} )
    #set( $expValues = {} )
    #set( $expSet = {} )
    #set( $expAdd = {} )
    #set( $expRemove = [] )

    ## Increment "version" by 1 **
    $!{expAdd.put("version", ":newVersion")}
    $!{expValues.put(":newVersion", { "N" : 1 })}

    ## Iterate through each argument, skipping "id" and "expectedVersion" **
    #foreach( $entry in $context.arguments.entrySet() )
        #if( $entry.key != "id" && $entry.key != "expectedVersion" )
            #if( (!$entry.value) && ("$!{entry.value}" == "") )
```

```
                ## If the argument is set to "null", then remove that attribute from
the item in DynamoDB **

                #set( $discard = ${expRemove.add("#${entry.key}")} )
                $!{expNames.put("#${entry.key}", "$entry.key")}
            #else
                ## Otherwise set (or update) the attribute on the item in DynamoDB **

                $!{expSet.put("#${entry.key}", ":${entry.key}")}
                $!{expNames.put("#${entry.key}", "$entry.key")}

                #if( $entry.key == "ups" || $entry.key == "downs" )
                    $!{expValues.put(":${entry.key}", { "N" : $entry.value })}
                #else
                    $!{expValues.put(":${entry.key}", { "S" : "${entry.value}" })}
                #end
            #end
        #end
    #end

    ## Start building the update expression, starting with attributes we're going to
SET **
    #set( $expression = "" )
    #if( !${expSet.isEmpty()} )
        #set( $expression = "SET" )
        #foreach( $entry in $expSet.entrySet() )
            #set( $expression = "${expression} ${entry.key} = ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Continue building the update expression, adding attributes we're going to ADD **
    #if( !${expAdd.isEmpty()} )
        #set( $expression = "${expression} ADD" )
        #foreach( $entry in $expAdd.entrySet() )
            #set( $expression = "${expression} ${entry.key} ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end
```

```
    ## Continue building the update expression, adding attributes we're going to REMOVE
 **
    #if( !${expRemove.isEmpty()} )
        #set( $expression = "${expression} REMOVE" )

        #foreach( $entry in $expRemove )
            #set( $expression = "${expression} ${entry}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Finally, write the update expression into the document, along with any
 expressionNames and expressionValues **
    "update" : {
        "expression" : "${expression}"
        #if( !${expNames.isEmpty()} )
            ,"expressionNames" : $utils.toJson($expNames)
        #end
        #if( !${expValues.isEmpty()} )
            ,"expressionValues" : $utils.toJson($expValues)
        #end
    },

    "condition" : {
        "expression"       : "version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" :
 $util.dynamodb.toDynamoDBJson($ctx.args.expectedVersion)
        }
    }
}
```

DynamoDB UpdateItem API の詳細については、「DynamoDB API のドキュメント」を参照してください。

DeleteItem

DeleteItem リクエストマッピングドキュメントでは、 AWS AppSync DynamoDB リゾルバーに DynamoDB へのDeleteItemリクエストを行うように指示し、以下を指定できます。

- DynamoDB の項目のキー

- 処理が成功する条件

DeleteItem マッピングドキュメントの構造は次のとおりです。

```
{
    "version" : "2018-05-29",
    "operation" : "DeleteItem",
    "customPartitionKey" : "foo",
    "populateIndexFields" : boolean value,
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "condition" : {
        ...
    },
    "_version" : 1
}
```

各フィールドの定義は以下のようになります。

DeleteItem フィールド

DeleteItem フィールドリスト

`version`

テンプレート定義バージョン `2017-02-28` と `2018-05-29` は現在サポートされています。この値は必須です。

`operation`

実行する DynamoDB の処理。`DeleteItem` DynamoDB の処理を実行するには、これを `DeleteItem` に設定する必要があります。この値は必須です。

`key`

DynamoDB の項目のキー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

condition

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件を指定していない場合、DeleteItem リクエストによって、現在の状態にかかわらず、項目が削除されます。条件の詳細については、「条件式」を参照してください。この値はオプションです。

_version

項目の既知の最新バージョンを表す数値。この値はオプションです。このフィールドは競合の検出に使用され、バージョン管理されたデータソースでのみサポートされます。

customPartitionKey

有効にすると、この文字列値は、バージョニングが有効になっているときにデルタ同期テーブルで使用される ds_sk および ds_pk レコードの形式を変更します (詳細については、「AWS AppSyncデベロッパーガイド」の「競合検出と同期」を参照)。有効にすると、populateIndexFields エントリの処理も有効になります。このフィールドはオプションです。

populateIndexFields

ブール値で、**customPartitionKey** と一緒に有効にすると、差分同期テーブル、具体的には gsi_ds_pk と gsi_ds_sk 列のレコードごとに新しいエントリが作成されます。詳細については、「AWS AppSyncデベロッパーガイド」の「競合検出と同期」を参照してください。このフィールドはオプションです。

DynamoDB から削除された項目が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、マッピングコンテキスト ($context.result) で参照できます。

DynamoDB の型変換の詳細については、「型システム (リクエストマッピング)」を参照してください。

レスポンスマッピングテンプレートの詳細については、「リゾルバーのマッピングテンプレートの概要」を参照してください。

例 1

以下は、GraphQL ミューテーション deleteItem(id: ID!) のマッピングテンプレートです。この ID に対応する項目がある場合は、削除されます。

```
{
```

```
    "version" : "2017-02-28",
    "operation" : "DeleteItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

例 2

以下は、GraphQL ミューテーション `deleteItem(id: ID!, expectedVersion: Int!)` のマッピングテンプレートです。この ID に対応する項目がある場合は、その `version` フィールドに `expectedVersion` が設定されているときにのみ削除されます。

```
{
    "version" : "2017-02-28",
    "operation" : "DeleteItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "condition" : {
        "expression"       : "attribute_not_exists(id) OR version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" : $util.dynamodb.toDynamoDBJson($expectedVersion)
        }
    }
}
```

DynamoDB `DeleteItem` API の詳細については、「DynamoDB API のドキュメント」を参照してください。

クエリ

`Query` リクエストマッピングドキュメントでは、 AWS AppSync DynamoDB リゾルバーに DynamoDB への`Query`リクエストを行うように指示し、以下を指定できます。

- キー式
- 使用するインデックス
- 任意の追加フィルタ
- 返す項目の数
- 整合性のある読み込みを使用するかどうか

- クエリの方向 (前方または後方)
- ページ分割トークン

Query マッピングドキュメントの構造は次のとおりです。

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "query" : {
        "expression" : "some expression",
        "expressionNames" : {
            "#foo" : "foo"
        },
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "index" : "fooIndex",
    "nextToken" : "a pagination token",
    "limit" : 10,
    "scanIndexForward" : true,
    "consistentRead" : false,
    "select" : "ALL_ATTRIBUTES" | "ALL_PROJECTED_ATTRIBUTES" | "SPECIFIC_ATTRIBUTES",
    "filter" : {
        ...
    },
    "projection" : {
        ...
    }
}
```

各フィールドの定義は以下のようになります。

Query フィールド

Query フィールドリスト

`version`

テンプレート定義バージョン `2017-02-28` と `2018-05-29` は現在サポートされています。この値は必須です。

operation

実行する DynamoDB の処理。Query DynamoDB の処理を実行するには、これを Query に設定する必要があります。この値は必須です。

query

query セクションには、DynamoDB から取得する項目を指示するキー条件式を指定することができます。キー条件式の記述方法の詳細については、DynamoDB KeyConditions のドキュメントを参照してください。このセクションの指定は必須です。

expression

クエリ式です。このフィールドの指定は必須です。

expressionNames

式の属性名のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される名前のプレースホルダーに対応し、値は DynamoDB の項目の属性名と一致する文字列でなければなりません。このフィールドはオプションであり、expression で使用される式の属性名のプレースホルダーのみを入力します。

expressionValues

式の属性値のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される値のプレースホルダーに対応し、値は型付き値でなければなりません。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。このフィールドはオプションであり、expression で使用される式の属性値のプレースホルダーのみを入力します。

filter

DynamoDB からの結果が返される前に、その結果をフィルタリングするために使用する追加フィルタです。フィルタの詳細については、「フィルタ」を参照してください。このフィールドはオプションです。

index

クエリを実行するインデックスの名前です。DynamoDB クエリの処理により、ハッシュキーのプライマリキーインデックスに加えて、ローカルセカンダリインデックスとグローバルセカンダリインデックスをスキャンできます。指定されると、DynamoDB が指定されたインデックスにクエリを実行します。省略すると、プライマリキーインデックスに対してクエリが実行されます。

nextToken

前のクエリを継続するためのページ分割トークンです。これは前のクエリから取得されます。このフィールドはオプションです。

limit

評価する項目の最大数 (一致する項目の数であるとは限りません)。このフィールドはオプションです。

scanIndexForward

クエリを前方と後方のどちらに実行するかを示すブール値です。このフィールドはオプションであり、デフォルトは `true` です。

consistentRead

DynamoDB にクエリを実行する際に整合性のある読み込みを使用するかどうかを示すブール値です。このフィールドはオプションであり、デフォルトは `false` です。

select

デフォルトでは、 AWS AppSync DynamoDB リゾルバーはインデックスに射影された属性のみを返します。より多くの属性が必要な場合に、このフィールドを設定できます。このフィールドはオプションです。サポートされている値には以下があります。

ALL_ATTRIBUTES

指定されたテーブルまたはインデックスのすべての項目の属性を返します。ローカルセカンダリインデックスに対してクエリを実行する場合、DynamoDB は、親のテーブルからインデックスの項目に一致したすべての項目をフェッチします。インデックスがすべての項目の属性を射影するように設定されている場合、すべてのデータはローカルセカンダリインデックスから取得されるため、フェッチは必要ありません。

ALL_PROJECTED_ATTRIBUTES

インデックスにクエリを実行する場合のみ使用できます。インデックスに投射されたすべての属性を取得します。インデックスがすべての属性を投射するように設定されている場合、この返り値は `ALL_ATTRIBUTES` を指定した場合と同等になります。

SPECIFIC_ATTRIBUTES

`projection` の `expression` にリストされている属性のみを返します。この戻り値は、`Select` の値を指定せずに `projection` の `expression` を指定するのと同じです。

projection

DynamoDB オペレーションから返される属性を指定するために使用されるプロジェクション。プロジェクションの詳細については、「プロジェクション」を参照してください。このフィールドはオプションです。

DynamoDB からの結果が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、マッピングコンテキスト (`$context.result`) で参照できます。

DynamoDB の型変換の詳細については、「型システム (リクエストマッピング)」を参照してください。

レスポンスマッピングテンプレートの詳細については、「リゾルバーのマッピングテンプレートの概要」を参照してください。

結果は以下の構造を持ちます。

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

各フィールドの定義は以下のようになります。

items

DynamoDB クエリで返された項目を含むリストです。

nextToken

さらに結果がある場合、`nextToken` には別のリクエストで使用できるページ分割トークンが含まれています。 AWS AppSync は、DynamoDB から返されるページ分割トークンを暗号化して難読化します。これにより、テーブルデータが誤って呼び出し元に漏えいされるのを防ぎます。また、これらのページ分割トークンは、異なるリゾルバー間では使用できないことにも注意してください。

scannedCount

フィルタ式 (ある場合) が適用される前に、クエリの条件式に一致した項目の数です。

例

次の例は、GraphQL クエリ `getPosts(owner: ID!)` のマッピングテンプレートです。

この例では、テーブルのグローバルセカンダリインデックスにクエリが実行され、指定した ID が所有するすべての投稿が返されます。

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "query" : {
        "expression" : "ownerId = :ownerId",
        "expressionValues" : {
            ":ownerId" : $util.dynamodb.toDynamoDBJson($context.arguments.owner)
        }
    },
    "index" : "owner-index"
}
```

DynamoDB `Query` API の詳細については、「DynamoDB API のドキュメント」を参照してください。

Scan

`Scan` リクエストマッピングドキュメントでは、 AWS AppSync DynamoDB リゾルバーに DynamoDB への`Scan`リクエストを行うように指示し、以下を指定できます。

- 結果を除外するフィルタ
- 使用するインデックス
- 返す項目の数
- 整合性のある読み込みを使用するかどうか
- ページ分割トークン
- 並列スキャン

`Scan` マッピングドキュメントの構造は次のとおりです。

```
{
    "version" : "2017-02-28",
```

```
    "operation" : "Scan",
    "index" : "fooIndex",
    "limit" : 10,
    "consistentRead" : false,
    "nextToken" : "aPaginationToken",
    "totalSegments" : 10,
    "segment" : 1,
    "filter" : {
        ...
    },
    "projection" : {
        ...
    }
}
```

各フィールドの定義は以下のようになります。

Scan フィールド

Scan フィールドリスト

`version`

テンプレート定義バージョン `2017-02-28` と `2018-05-29` は現在サポートされています。この値は必須です。

`operation`

実行する DynamoDB の処理。`Scan` DynamoDB の処理を実行するには、これを `Scan` に設定する必要があります。この値は必須です。

`filter`

DynamoDB からの結果が返される前に、その結果をフィルタリングするために使用するフィルタです。フィルタの詳細については、「フィルタ」を参照してください。このフィールドはオプションです。

`index`

クエリを実行するインデックスの名前です。DynamoDB クエリの処理により、ハッシュキーのプライマリキーインデックスに加えて、ローカルセカンダリインデックスとグローバルセカンダリインデックスをスキャンできます。指定されると、DynamoDB が指定されたインデックスにクエリを実行します。省略すると、プライマリキーインデックスに対してクエリが実行されます。

limit

一度に評価する項目の最大数です。このフィールドはオプションです。

consistentRead

DynamoDB にクエリを実行する際に整合性のある読み込みを使用するかどうかを示すブール値です。このフィールドはオプションであり、デフォルトは `false` です。

nextToken

前のクエリを継続するためのページ分割トークンです。これは前のクエリから取得されます。このフィールドはオプションです。

select

デフォルトでは、 AWS AppSync DynamoDB リゾルバーはインデックスに射影される属性のみを返します。より多くの属性が必要な場合にこのフィールドを設定します。このフィールドはオプションです。サポートされている値には以下があります。

ALL_ATTRIBUTES

指定されたテーブルまたはインデックスのすべての項目の属性を返します。ローカルセカンダリインデックスに対してクエリを実行する場合、DynamoDB は、親のテーブルからインデックスの項目に一致したすべての項目をフェッチします。インデックスがすべての項目の属性を射影するように設定されている場合、すべてのデータはローカルセカンダリインデックスから取得されるため、フェッチは必要ありません。

ALL_PROJECTED_ATTRIBUTES

インデックスにクエリを実行する場合のみ使用できます。インデックスに投射されたすべての属性を取得します。インデックスがすべての属性を投射するように設定されている場合、この返り値は `ALL_ATTRIBUTES` を指定した場合と同等になります。

SPECIFIC_ATTRIBUTES

`projection` の `expression` にリストされている属性のみを返します。この戻り値は、`Select` の値を指定せずに `projection` の `expression` を指定するのと同じです。

totalSegments

並列スキャンが実行されるまでにテーブルを分割するセグメントの数です。このフィールドはオプションですが、`segment` を指定した場合には指定する必要があります。

segment

並列スキャンの実行時のこの処理でのテーブルセグメントです。このフィールドはオプションですが、totalSegments を指定した場合には指定する必要があります。

projection

DynamoDB オペレーションから返される属性を指定するために使用されるプロジェクション。プロジェクションの詳細については、「プロジェクション」を参照してください。このフィールドはオプションです。

DynamoDB スキャンにより返された結果が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、マッピングコンテキスト ($context.result) で参照できます。

DynamoDB の型変換の詳細については、「型システム (リクエストマッピング)」を参照してください。

レスポンスマッピングテンプレートの詳細については、「リゾルバーのマッピングテンプレートの概要」を参照してください。

結果は以下の構造を持ちます。

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

各フィールドの定義は以下のようになります。

items

DynamoDB スキャンにより返された項目を含むリストです。

nextToken

さらに結果がある場合、 には別のリクエストで使用できるページ分割トークンnextTokenが含まれています。 AWS AppSync は、DynamoDB から返されたページ分割トークンを暗号化して難読化します。これにより、テーブルデータが誤って呼び出し元に漏えいされるのを防ぎます。また、これらのページ分割トークンは異なるリゾルバー間では使用できません。

scannedCount

フィルタ式 (ある場合) が適用される前に、DynamoDB により取得された項目の数です。

例 1

次の例は、GraphQL クエリ: `allPosts` のマッピングテンプレートです。

この例では、テーブル内のすべてのエントリが返されます。

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
}
```

例 2

次の例は、GraphQL クエリ: `postsMatching(title: String!)` のマッピングテンプレートです。

この例では、タイトルが `title` 引数で始まるテーブル内のすべてのエントリが返されます。

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "filter" : {
        "expression" : "begins_with(title, :title)",
        "expressionValues" : {
            ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
        },
    }
}
```

DynamoDB Scan API の詳細については、「DynamoDB API のドキュメント」を参照してください。

同期

Sync リクエストマッピングドキュメントを使用すると、DynamoDB テーブルからすべての結果を取得し、最後のクエリ (差分更新) 以降に変更されたデータのみを受け取ることができます。 Syncリ

クエストは、バージョン管理された DynamoDB データソースに対してのみ実行できます。以下を指定することができます。

- 結果を除外するフィルタ
- 返す項目の数
- ページ分割トークン
- 最後の Sync オペレーションが開始された日時

Sync マッピングドキュメントの構造は次のとおりです。

```
{
    "version" : "2018-05-29",
    "operation" : "Sync",
    "basePartitionKey": "Base Tables PartitionKey",
    "deltaIndexName": "delta-index-name",
    "limit" : 10,
    "nextToken" : "aPaginationToken",
    "lastSync" :  1550000000000,
    "filter" : {
        ...
    }
}
```

各フィールドの定義は以下のようになります。

Sync フィールド

Sync フィールドリスト

`version`

テンプレート定義のバージョンです。現在、`2018-05-29` のみがサポートされています。この値は必須です。

`operation`

実行する DynamoDB の処理。Sync の処理を実行するには、これに Sync を設定する必要があります。この値は必須です。

filter

DynamoDB からの結果が返される前に、その結果をフィルタリングするために使用するフィルタです。フィルタの詳細については、「フィルタ」を参照してください。このフィールドはオプションです。

limit

一度に評価する項目の最大数です。このフィールドはオプションです。省略した場合、デフォルトの制限は 100 項目に設定されます。このフィールドの最大値は 1000 項目です。

nextToken

前のクエリを継続するためのページ分割トークンです。これは前のクエリから取得されます。このフィールドはオプションです。

lastSync

最後に成功した Sync オペレーションが開始されたエポックミリ秒単位の時刻。指定すると、lastSync 以降に変更された項目のみが返されます。このフィールドはオプションです。最初の Sync オペレーションからすべてのページを取得した後にのみ入力する必要があります。省略した場合は、ベーステーブルの結果が返されます。それ以外の場合は、差分テーブルの結果が返されます。

basePartitionKey

Sync オペレーションを実行する際に使用されるベーステーブルのパーティションキー。このフィールドにより、テーブルがカスタムパーティションキーを使用している場合に Sync オペレーションを実行できます。これはオプションのフィールドです。

deltaIndexName

Sync オペレーションに使用されるインデックス。このインデックスは、テーブルがカスタムパーティションキーを使用する場合に、デルタストアテーブル全体で Sync オペレーション作を有効にするために必要です。Sync オペレーションは GSI (gsi_ds_pk と gsi_ds_sk で作成) 上で実行されます。このフィールドはオプションです。

DynamoDB 同期により返された結果が自動的に GraphQL プリミティブ型と JSON プリミティブ型に変換され、マッピングコンテキスト ($context.result) で参照できます。

DynamoDB の型変換の詳細については、「型システム (リクエストマッピング)」を参照してください。

レスポンスマッピングテンプレートの詳細については、「リゾルバーのマッピングテンプレートの概要」を参照してください。

結果は以下の構造を持ちます。

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10,
    startedAt = 1550000000000
}
```

各フィールドの定義は以下のようになります。

items

同期により返された項目を含むリストです。

nextToken

さらに結果がある場合、 には別のリクエストで使用できるページ分割トークンnextTokenが含まれています。 AWS AppSync は、DynamoDB から返されたページ分割トークンを暗号化して難読化します。これにより、テーブルデータが誤って呼び出し元に漏えいされるのを防ぎます。また、これらのページ分割トークンは異なるリゾルバー間では使用できません。

scannedCount

フィルタ式 (ある場合) が適用される前に、DynamoDB により取得された項目の数です。

startedAt

エポックミリ秒単位の時刻ですが、同期オペレーションが開始されれば、ローカルに保存して別のリクエストで `lastSync` の引数として使用することができます。ページ分割トークンがリクエストに含まれている場合、この値は、結果の最初のページのリクエストによって返されたものと同じになります。

例

次の例は、GraphQL クエリ: `syncPosts(nextToken: String, lastSync: AWSTimestamp)` のマッピングテンプレートです。

この例では、lastSync を省略すると、ベーステーブルのすべてのエントリが返されます。lastSync が指定されている場合は、lastSync 以降に変更された差分同期テーブルのエントリのみが返されます。

```
{
    "version" : "2018-05-29",
    "operation" : "Sync",
    "limit": 100,
    "nextToken": $util.toJson($util.defaultIfNull($ctx.args.nextToken, null)),
    "lastSync": $util.toJson($util.defaultIfNull($ctx.args.lastSync, null))
}
```

BatchGetItem

BatchGetItem リクエストマッピングドキュメントを使用すると、 AWS AppSync DynamoDB リゾルバーに、複数のテーブルにまたがる可能性のある複数の項目を取得するリクエストBatchGetItemを DynamoDB に行うように指示できます。このリクエストテンプレートでは、以下の情報を指定する必要があります。

- 項目を取得するテーブルの名前
- 各テーブルから取得する項目のキー

DynamoDB の BatchGetItem 制限が適用されるため、条件式を指定することはできません。

BatchGetItem マッピングドキュメントの構造は次のとおりです。

```
{
    "version" : "2018-05-29",
    "operation" : "BatchGetItem",
    "tables" : {
        "table1": {
           "keys": [
              ## Item to retrieve Key
              {
                   "foo" : ... typed value,
                   "bar" : ... typed value
              },
              ## Item2 to retrieve Key
              {
                   "foo" : ... typed value,
```

```
                    "bar" : ... typed value
                }
            ],
            "consistentRead": true|false,
            "projection" : {
                ...
            }
        },
        "table2": {
           "keys": [
             ## Item3 to retrieve Key
             {
                  "foo" : ... typed value,
                  "bar" : ... typed value
             },
             ## Item4 to retrieve Key
             {
                  "foo" : ... typed value,
                  "bar" : ... typed value
             }
            ],
            "consistentRead": true|false,
            "projection" : {
                ...
            }
        }
    }
}
```

各フィールドの定義は以下のようになります。

BatchGetItem フィールド

BatchGetItem フィールドリスト

`version`

テンプレート定義のバージョンです。2018-05-29 のみサポートされています。この値は必須です。

`operation`

実行する DynamoDB の処理。`BatchGetItem` DynamoDB の処理を実行するには、これを `BatchGetItem` に設定する必要があります。この値は必須です。

tables

項目を取得する DynamoDB テーブル。値は、キーとしてテーブル名が指定されているマップです。1 つ以上のテーブルを指定する必要があります。この `tables` の値は必須です。

keys

取り出す項目のプライマリキーを表す DynamoDB キーのリスト。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。

consistentRead

GetItem 処理の実行時に整合性のある読み込みを使用するかどうか。この値はオプションであり、デフォルトは false です。

projection

DynamoDB オペレーションから返される属性を指定するために使用されるプロジェクション。プロジェクションの詳細については、「プロジェクション」を参照してください。このフィールドはオプションです。

覚えておくべきポイント:

- 項目がテーブルから取得されなかった場合は、そのテーブルの data ブロックに null 要素があります。
- 呼び出し結果は、リクエストマッピングテンプレート内で提供された順序に基づいて、テーブル別にソートされます。
- `Get` 内の各 `BatchGetItem` コマンドはアトミックですが、バッチは部分的に処理される場合があります。エラーのためにバッチが部分的に処理された場合、未処理のキーは unprocessedKeys ブロック内に呼び出し結果の一部として返されます。
- `BatchGetItem` は 100 キーに制限されています。

以下に示しているのは、リクエストマッピングテンプレートの例です。

```
{
  "version": "2018-05-29",
  "operation": "BatchGetItem",
  "tables": {
```

```
    "authors": [
        {
          "author_id": {
            "S": "a1"
          }
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          }
        }
    ],
  }
}
```

$ctx.result で使用可能な呼び出し結果は以下のとおりです。

```
{
   "data": {
     "authors": [null],
     "posts": [
        # Was retrieved
        {
          "author_id": "a1",
          "post_id": "p2",
          "post_title": "title",
          "post_description": "description",
        }
     ]
   },
   "unprocessedKeys": {
     "authors": [
        # This item was not processed due to an error
        {
          "author_id": "a1"
        }
      ],
     "posts": []
```

```
    }
}
```

$ctx.error にエラーに関する詳細が含まれています。data キー、unprocessedKeys キー、およびリクエストマッピングテンプレートで渡された各テーブルキーは呼び出し結果に必ずあります。削除された項目は data ブロックにあります。処理されなかった項目は、data ブロック内で null としてマークされ、unprocessedKeys ブロックに挿入されます。

より完全な例については、AppSync の DynamoDB Batch の「チュートリアル: DynamoDB Batch リゾルバー」を参照してください。

BatchDeleteItem

BatchDeleteItem リクエストマッピングドキュメントを使用すると、 AWS AppSync DynamoDB リゾルバーに、複数のテーブルにまたがる可能性のある複数の項目を削除するリクエストを DynamoDB BatchWriteItemに行うように指示できます。このリクエストテンプレートでは、以下の情報を指定する必要があります。

- 項目を削除するテーブルの名前
- 各テーブルから削除する項目のキー

DynamoDB の BatchWriteItem 制限が適用されるため、条件式を指定することはできません。

BatchDeleteItem マッピングドキュメントの構造は次のとおりです。

```
{
    "version" : "2018-05-29",
    "operation" : "BatchDeleteItem",
    "tables" : {
        "table1": [
        ## Item to delete Key
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        },
        ## Item2 to delete Key
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        }],
```

```
        "table2": [
        ## Item3 to delete Key
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        },
        ## Item4 to delete Key
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        }],
    }
}
```

各フィールドの定義は以下のようになります。

BatchDeleteItem フィールド

BatchDeleteItem フィールドリスト

`version`

テンプレート定義のバージョンです。`2018-05-29` のみサポートされています。この値は必須です。

`operation`

実行する DynamoDB の処理。`BatchDeleteItem` DynamoDB の処理を実行するには、これを `BatchDeleteItem` に設定する必要があります。この値は必須です。

`tables`

項目を削除する DynamoDB テーブル。各テーブルは、削除する項目のプライマリキーを表す DynamoDB キーのリストです。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。1 つ以上のテーブルを指定する必要があります。値 が必要です。

覚えておくべきポイント:

- `DeleteItem` オペレーションとは対照的に、完全に削除された項目はレスポンスで返されません。渡されたキーのみが返されます。

- 項目がテーブルから削除されなかった場合は、そのテーブルの data ブロックに null 要素があります。
- 呼び出し結果は、リクエストマッピングテンプレート内で提供された順序に基づいて、テーブル別にソートされます。
- BatchDeleteItem 内部の各 Delete コマンドはアトミックです。ただし、バッチは部分的に処理できます。エラーのためにバッチが部分的に処理された場合、未処理のキーは unprocessedKeys ブロック内に呼び出し結果の一部として返されます。
- BatchDeleteItem は 25 キーに制限されています。
- このオペレーション、競合検出で使用するとサポートされていません。両方を同時に使用すると、エラーが発生する可能性があります。

以下に示しているのは、リクエストマッピングテンプレートの例です。

```
{
  "version": "2018-05-29",
  "operation": "BatchDeleteItem",
  "tables": {
    "authors": [
        {
          "author_id": {
            "S": "a1"
          }
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          }
        }
    ],
  }
}
```

$ctx.result で使用可能な呼び出し結果は以下のとおりです。

```
{
```

```
   "data": {
     "authors": [null],
     "posts": [
        # Was deleted
        {
          "author_id": "a1",
          "post_id": "p2"
        }
     ]
   },
   "unprocessedKeys": {
     "authors": [
        # This key was not processed due to an error
        {
          "author_id": "a1"
        }
      ],
     "posts": []
   }
}
```

$ctx.error にエラーに関する詳細が含まれています。data キー、unprocessedKeys キー、およびリクエストマッピングテンプレートで渡された各テーブルキーは呼び出し結果に必ずあります。削除された項目は data ブロックにあります。処理されなかった項目は、data ブロック内で null としてマークされ、unprocessedKeys ブロックに挿入されます。

より完全な例については、AppSync の DynamoDB Batch の「チュートリアル: DynamoDB Batch リゾルバー」を参照してください。

BatchPutItem

BatchPutItem リクエストマッピングドキュメントを使用すると、 AWS AppSync DynamoDB リゾルバーに、複数のテーブルにまたがる可能性のある複数の項目を配置するように DynamoDB にBatchWriteItemリクエストするように指示できます。このリクエストテンプレートでは、以下の情報を指定する必要があります。

- 項目を挿入するテーブルの名前
- 各テーブルに挿入するすべての項目

DynamoDB の BatchWriteItem 制限が適用されるため、条件式を指定することはできません。

BatchPutItem マッピングドキュメントの構造は次のとおりです。

```
{
    "version" : "2018-05-29",
    "operation" : "BatchPutItem",
    "tables" : {
        "table1": [
        ## Item to put
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        },
        ## Item2 to put
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        }],
        "table2": [
        ## Item3 to put
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        },
        ## Item4 to put
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        }],
    }
}
```

各フィールドの定義は以下のようになります。

BatchPutItem フィールド

BatchPutItem フィールドリスト

version

テンプレート定義のバージョンです。2018-05-29 のみサポートされています。この値は必須です。

operation

実行する DynamoDB の処理。BatchPutItem DynamoDB の処理を実行するには、これを BatchPutItem に設定する必要があります。この値は必須です。

tables

項目を挿入する DynamoDB テーブル。各テーブルエントリは、この特定のテーブルに挿入する DynamoDB 項目のリストを表します。1 つ以上のテーブルを指定する必要があります。この値は必須です。

覚えておくべきポイント:

- 完全に挿入された項目がレスポンスに返されます (正常に挿入された場合)。
- 項目がテーブルに挿入されなかった場合は、そのテーブルの data ブロックに null 要素があります。
- 挿入された項目は、リクエストマッピングテンプレート内で提供された順序に基づいて、テーブル別にソートされます。
- BatchPutItem 内の各 Put コマンドはアトミックですが、バッチは部分的に処理される場合があります。エラーのためにバッチが部分的に処理された場合、未処理のキーは unprocessedKeys ブロック内に呼び出し結果の一部として返されます。
- BatchPutItem は 25 項目に制限されています。
- このオペレーション、競合検出で使用するとサポートされていません。両方を同時に使用すると、エラーが発生する可能性があります。

以下に示しているのは、リクエストマッピングテンプレートの例です。

```
{
  "version": "2018-05-29",
  "operation": "BatchPutItem",
  "tables": {
    "authors": [
        {
          "author_id": {
            "S": "a1"
          },
          "author_name": {
            "S": "a1_name"
          }
```

```
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          },
          "post_title": {
            "S": "title"
          }
        }
    ],
  }
}
```

$ctx.result で使用可能な呼び出し結果は以下のとおりです。

```
{
   "data": {
     "authors": [
         null
     ],
     "posts": [
        # Was inserted
        {
          "author_id": "a1",
          "post_id": "p2",
          "post_title": "title"
        }
     ]
   },
   "unprocessedItems": {
     "authors": [
        # This item was not processed due to an error
        {
          "author_id": "a1",
          "author_name": "a1_name"
        }
      ],
     "posts": []
```

```
    }
}
```

$ctx.error にエラーに関する詳細が含まれています。data キー、The keys data, unprocessedItems キー、およびリクエストマッピングテンプレートで渡された各テーブルキーは呼び出し結果に必ずあります。挿入された項目は data ブロックにあります。処理されなかった項目は、data ブロック内で null としてマークされ、unprocessedItems ブロックに挿入されます。

より完全な例については、AppSync の DynamoDB Batch の「チュートリアル: DynamoDB Batch リゾルバー」を参照してください。

TransactGetItems

TransactGetItems リクエストマッピングドキュメントを使用すると、 AWS AppSync DynamoDB リゾルバーに、複数のテーブルにまたがる可能性のある複数の項目を取得するリクエストを DynamoDB TransactGetItemsに行うように指示できます。このリクエストテンプレートでは、以下の情報を指定する必要があります。

- 項目の取得元となる各リクエスト項目のテーブル名
- 各テーブルから取得する各リクエスト項目のキー

DynamoDB の TransactGetItems 制限が適用されるため、条件式を指定することはできません。

TransactGetItems マッピングドキュメントの構造は次のとおりです。

```
{
    "version": "2018-05-29",
    "operation": "TransactGetItems",
    "transactItems": [
       ## First request item
       {
           "table": "table1",
           "key": {
               "foo": ... typed value,
               "bar": ... typed value
           },
           "projection" : {
                ...
           }
       },
```

```
        ## Second request item
        {
            "table": "table2",
            "key": {
                "foo": ... typed value,
                "bar": ... typed value
            },
            "projection" : {
                 ...
            }
        }
    ]
}
```

各フィールドの定義は以下のようになります。

TransactGetItems フィールド

TransactGetItems フィールドリスト

`version`

テンプレート定義のバージョンです。`2018-05-29` のみサポートされています。この値は必須です。

`operation`

実行する DynamoDB の処理。`TransactGetItems` DynamoDB の処理を実行するには、これを `TransactGetItems` に設定する必要があります。この値は必須です。

`transactItems`

含めるリクエスト項目。値はリクエスト項目の配列です。少なくとも 1 つのリクエスト項目を指定する必要があります。この `transactItems` の値は必須です。

`table`

項目の取得元となる DynamoDB テーブル。値はテーブル名の文字列です。この `table` の値は必須です。

`key`

取り出す項目のプライマリキーを表す DynamoDB キー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造に

よって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。

`projection`

DynamoDB オペレーションから返される属性を指定するために使用されるプロジェクション。プロジェクションの詳細については、「プロジェクション」を参照してください。このフィールドはオプションです。

覚えておくべきポイント:

- トランザクションが成功すると、`items` ブロック内で取得された項目の順序はリクエスト項目の順序と同じになります。
- トランザクションは、オールオアナッシング方式で実行されます。いずれかのリクエスト項目でエラーが発生した場合、トランザクション全体は実行されず、エラーの詳細が返されます。
- リクエスト項目を取得できなくても、エラーではありません。代わりに、null要素が対応する位置の項目ブロックに表示されます。
- トランザクションのエラーが TransactionCanceledException である場合、`cancellationReasons` ブロックに入力されます。`cancellationReasons` ブロック内のキャンセル理由の順序は、リクエスト項目の順序と同じになります。
- `TransactGetItems` のリクエスト項目数は 100 個に制限されています。

以下に示しているのは、リクエストマッピングテンプレートの例です。

```
{
    "version": "2018-05-29",
    "operation": "TransactGetItems",
    "transactItems": [
       ## First request item
       {
           "table": "posts",
           "key": {
               "post_id": {
                 "S": "p1"
               }
           }
       },
       ## Second request item
       {
```

```
            "table": "authors",
            "key": {
                "author_id": {
                  "S": a1
                }
            }
        }
    ]
}
```

トランザクションが成功し、最初にリクエストされた項目だけが取得された場合、`$ctx.result` で使用できる呼び出し結果は次のようになります。

```
{
    "items": [
       {
           // Attributes of the first requested item
           "post_id": "p1",
           "post_title": "title",
           "post_description": "description"
       },
       // Could not retrieve the second requested item
       null,
    ],
    "cancellationReasons": null
}
```

最初のリクエスト項目によって発生した TransactionCanceledException が原因でトランザクションが失敗した場合、`$ctx.result` で使用可能な呼び出し結果は次のようになります。

```
{
    "items": null,
    "cancellationReasons": [
       {
           "type":"Sample error type",
           "message":"Sample error message"
       },
       {
           "type":"None",
           "message":"None"
       }
    ]
```

```
}
```

$ctx.error にエラーに関する詳細が含まれています。キー items と cancellationReasons は、$ctx.result にあることが保証されています。

より完全な例については、AppSync を使用した DynamoDB トランザクションのチュートリアル「チュートリアル: DynamoDB Transaction リゾルバー」を参照してください。

TransactWriteItems

TransactWriteItems リクエストマッピングドキュメントでは、 AWS AppSync DynamoDB リゾルバーに、複数のテーブルに複数の項目を書き込むように DynamoDB にTransactWriteItemsリクエストするように指示できます。このリクエストテンプレートでは、以下の情報を指定する必要があります。

- 各リクエスト項目の書き込み先テーブル名
- 実行する各リクエスト項目のオペレーション。サポートされているオペレーションには、PutItem、UpdateItem、DeleteItem、および ConditionCheck の 4 種類があります。
- 書き込む各リクエスト項目のキー

DynamoDB の TransactWriteItems 制限が適用されます。

TransactWriteItems マッピングドキュメントの構造は次のとおりです。

```
{
    "version": "2018-05-29",
    "operation": "TransactWriteItems",
    "transactItems": [
       {
           "table": "table1",
           "operation": "PutItem",
           "key": {
               "foo": ... typed value,
               "bar": ... typed value
           },
           "attributeValues": {
               "baz": ... typed value
           },
           "condition": {
               "expression": "someExpression",
               "expressionNames": {
```

```
                "#foo": "foo"
            },
            "expressionValues": {
                ":bar": ... typed value
            },
            "returnValuesOnConditionCheckFailure": true|false
        }
    },
    {
        "table":"table2",
        "operation": "UpdateItem",
        "key": {
            "foo": ... typed value,
            "bar": ... typed value
        },
        "update": {
            "expression": "someExpression",
            "expressionNames": {
                "#foo": "foo"
            },
            "expressionValues": {
                ":bar": ... typed value
            }
        },
        "condition": {
            "expression": "someExpression",
            "expressionNames": {
                "#foo":"foo"
            },
            "expressionValues": {
                ":bar": ... typed value
            },
            "returnValuesOnConditionCheckFailure": true|false
        }
    },
    {
        "table": "table3",
        "operation": "DeleteItem",
        "key":{
            "foo": ... typed value,
            "bar": ... typed value
        },
        "condition":{
            "expression": "someExpression",
```

```
                "expressionNames": {
                    "#foo": "foo"
                },
                "expressionValues": {
                    ":bar": ... typed value
                },
                "returnValuesOnConditionCheckFailure": true|false
            }
        },
        {
            "table": "table4",
            "operation": "ConditionCheck",
            "key":{
                "foo": ... typed value,
                "bar": ... typed value
            },
            "condition":{
                "expression": "someExpression",
                "expressionNames": {
                    "#foo": "foo"
                },
                "expressionValues": {
                    ":bar": ... typed value
                },
                "returnValuesOnConditionCheckFailure": true|false
            }
        }
    ]
}
```

TransactWriteItems フィールド

TransactWriteItems フィールドリスト

各フィールドの定義は以下のようになります。

`version`

テンプレート定義のバージョンです。`2018-05-29` のみサポートされています。この値は必須です。

operation

実行する DynamoDB の処理。`TransactWriteItems` DynamoDB の処理を実行するには、これを `TransactWriteItems` に設定する必要があります。この値は必須です。

transactItems

含めるリクエスト項目。値はリクエスト項目の配列です。少なくとも 1 つのリクエスト項目を指定する必要があります。この `transactItems` の値は必須です。

`PutItem` の場合、各フィールドの定義は以下のようになります。

table

送信先のの DynamoDB テーブル。値はテーブル名の文字列です。この `table` の値は必須です。

operation

実行する DynamoDB の処理。`PutItem` DynamoDB の処理を実行するには、これを `PutItem` に設定する必要があります。この値は必須です。

key

配置する項目のプライマリキーを表す DynamoDB キー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

attributeValues

DynamoDB に渡す項目の残りの属性です。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。このフィールドはオプションです。

condition

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件が指定されていない場合、`PutItem` リクエストはその項目の既存のエントリを上書きします。条件チェックに失敗した場合に、既存の項目を取得し直すかどうかを指定できます。トランザクション条件の詳細については、「トランザクション条件式」を参照してください。この値はオプションです。

`UpdateItem` の場合、各フィールドの定義は以下のようになります。

table

更新する DynamoDB テーブル。値はテーブル名の文字列です。この `table` の値は必須です。

operation

実行する DynamoDB の処理。`UpdateItem` DynamoDB の処理を実行するには、これを `UpdateItem` に設定する必要があります。この値は必須です。

key

更新する項目のプライマリキーを表す DynamoDB キー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

update

`update` セクションには、 DynamoDBの項目の更新方法を示す更新式を指定することができます。更新式の記述方法の詳細については、「DynamoDB UpdateExpressions」のドキュメントを参照してください。このセクションは必須です。

condition

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件を指定していない場合は、`UpdateItem` リクエストによって、現在の状態にかかわらず、既存のエントリが更新されます。条件チェックに失敗した場合に、既存の項目を取得し直すかどうかを指定できます。トランザクション条件の詳細については、「トランザクション条件式」を参照してください。この値はオプションです。

`DeleteItem` の場合、各フィールドの定義は以下のようになります。

table

項目を削除する DynamoDB テーブル。値はテーブル名の文字列です。この `table` の値は必須です。

operation

実行する DynamoDB の処理。`DeleteItem` DynamoDB の処理を実行するには、これを `DeleteItem` に設定する必要があります。この値は必須です。

key

削除する項目のプライマリキーを表す DynamoDB キー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

condition

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件を指定していない場合、DeleteItem リクエストによって、現在の状態にかかわらず、項目が削除されます。条件チェックに失敗した場合に、既存の項目を取得し直すかどうかを指定できます。トランザクション条件の詳細については、「トランザクション条件式」を参照してください。この値はオプションです。

ConditionCheck の場合、各フィールドの定義は以下のようになります。

table

条件をチェックする DynamoDB テーブル。値はテーブル名の文字列です。この table の値は必須です。

operation

実行する DynamoDB の処理。ConditionCheck DynamoDB の処理を実行するには、これを ConditionCheck に設定する必要があります。この値は必須です。

key

条件チェックする項目のプライマリキーを表す DynamoDB キー。DynamoDB の項目には、単一のハッシュキー、またはハッシュキーとソートキーが含まれています。これはテーブルの構造によって変わります。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この値は必須です。

condition

DynamoDB 内に既に存在するオブジェクトの状態に基づき、リクエストが成功するかどうかを判断する条件です。条件チェックに失敗した場合に、既存の項目を取得し直すかどうかを指定できます。トランザクション条件の詳細については、「トランザクション条件式」を参照してください。この値は必須です。

覚えておくべきポイント:

- リクエスト項目のキーのみがレスポンスに返されます (正常に挿入された場合)。キーの順序は、リクエスト項目の順序と同じです。
- トランザクションは、オールオアナッシング方式で実行されます。いずれかのリクエスト項目でエラーが発生した場合、トランザクション全体は実行されず、エラーの詳細が返されます。
- 同じ項目をターゲットにできるリクエスト項目が 2 つありません。それ以外の場合は、TransactionCanceledException エラーが発生します。
- トランザクションのエラーが TransactionCanceledException である場合、`cancellationReasons` ブロックに入力されます。リクエスト項目の条件チェックが失敗し、かつ `returnValuesOnConditionCheckFailure` を `false` に指定しなかった場合、テーブルに存在する項目が取得され、`item` で `cancellationReasons` ブロックの対応する位置に保存されます。
- `TransactWriteItems` のリクエスト項目数は 100 個に制限されています。
- このオペレーション、競合検出で使用するとサポートされていません。両方を同時に使用すると、エラーが発生する可能性があります。

以下に示しているのは、リクエストマッピングテンプレートの例です。

```
{
    "version": "2018-05-29",
    "operation": "TransactWriteItems",
    "transactItems": [
       {
           "table": "posts",
           "operation": "PutItem",
           "key": {
               "post_id": {
                   "S": "p1"
               }
           },
           "attributeValues": {
               "post_title": {
                   "S": "New title"
               },
               "post_description": {
                   "S": "New description"
               }
           },
           "condition": {
               "expression": "post_title = :post_title",
```

```
                "expressionValues": {
                    ":post_title": {
                        "S": "Expected old title"
                    }
                }
            }
        },
        {
            "table":"authors",
            "operation": "UpdateItem",
            "key": {
                "author_id": {
                    "S": "a1"
                },
            },
            "update": {
                "expression": "SET author_name = :author_name",
                "expressionValues": {
                    ":author_name": {
                        "S": "New name"
                    }
                }
            },
        }
    ]
}
```

トランザクションが成功した場合、$ctx.result で使用できる呼び出し結果は次のようになります。

```
{
    "keys": [
       // Key of the PutItem request
       {
           "post_id": "p1",
       },
       // Key of the UpdateItem request
       {
           "author_id": "a1"
       }
    ],
    "cancellationReasons": null
}
```

PutItem リクエストの条件チェックに失敗したためにトランザクションが失敗した場合、 で使用できる呼び出し結果は次のようになります。

```
{
    "keys": null,
    "cancellationReasons": [
       {
           "item": {
               "post_id": "p1",
               "post_title": "Actual old title",
               "post_description": "Old description"
           },
           "type": "ConditionCheckFailed",
           "message": "The condition check failed."
       },
       {
           "type": "None",
           "message": "None"
       }
    ]
}
```

$ctx.error にエラーに関する詳細が含まれています。keys および cancellationReasons が $ctx.result に存在することが保証されています。

より完全な例については、AppSync を使用した DynamoDB トランザクションのチュートリアル「チュートリアル: DynamoDB Transaction リゾルバー」を参照してください。

型システム (リクエストマッピング)

AWS AppSync DynamoDB リゾルバーを使用して DynamoDB テーブルを呼び出す場合、 AWS AppSync はその呼び出しで使用する各値のタイプを知る必要があります。これは、DynamoDB が GraphQL や JSON よりも多くのタイプのプリミティブ (セットやバイナリデータなど) をサポートしているためです。 AWS AppSync は、GraphQL と DynamoDB の間で変換するときにいくつかのヒントを必要とします。それ以外の場合は、テーブル内のデータの構造についていくつかの前提を行う必要があります。

DynamoDB のデータ型の詳細については、DynamoDB の「データ型記述子」および「データ型」の各ドキュメントを参照してください。

DynamoDB の値は、単一のキーと値のペアを含む JSON オブジェクトで表されます。キーは DynamoDB の型を指定し、値はその値自身を指定します。次の例では、キー S は値が文字列であることを示し、値 identifier がその文字列値です。

```
{ "S" : "identifier" }
```

JSON オブジェクトは複数のキーと値のペアを持つことはできません。複数のキーと値のペアが指定されている場合、リクエストマッピングドキュメントは解析されません。

DynamoDB の値は、値を指定する必要がある場合にはリクエストマッピングドキュメントのどこかで使用されます。これが必要になる箇所には、key セクションと attributeValue セクション、および式セクションの expressionValues セクションが含まれています。次の例では、DynamoDB の文字列値 identifier が、(おそらくは GetItem リクエストマッピングドキュメントの) key セクションの id フィールドに割り当てられています。

```
"key" : {
   "id" : { "S" : "identifier" }
}
```

サポートされているタイプ

AWS AppSync は、次の DynamoDB スカラー、ドキュメント、およびセットタイプをサポートしています。

文字列型 **S**

単一の文字列値です。DynamoDB の文字列値は次のように表されます。

```
{ "S" : "some string" }
```

以下は使用例です。

```
"key" : {
   "id" : { "S" : "some string" }
}
```

文字列セット型 **SS**

1 組の文字列値です。DynamoDB の文字列セット値は次のように表されます。

```
{ "SS" : [ "first value", "second value", ... ] }
```

以下は使用例です。

```
"attributeValues" : {
   "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
}
```

数値型 **N**

単一の数値です。DynamoDB の数値は次のように表されます。

```
{ "N" : 1234 }
```

以下は使用例です。

```
"expressionValues" : {
   ":expectedVersion" : { "N" : 1 }
}
```

数値セット型 **NS**

1 組の数値です。DynamoDB の数値セット値は次のように表されます。

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

以下は使用例です。

```
"attributeValues" : {
   "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }
}
```

バイナリ型 **B**

バイナリ値です。DynamoDB のバイナリ値は次のように表されます。

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

値は実際には文字列であり、文字列はバイナリデータの base64 でエンコードされた表現です。 AWS AppSync はこの文字列をバイナリ値にデコードしてから DynamoDB に送信します。 AWS AppSync は、RFC 2045 で定義されている base64 デコードスキームを使用します。base64 アルファベットにない文字は無視されます。

以下は使用例です。

```
"attributeValues" : {
   "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

バイナリセット型 **BS**

1 組のバイナリ値です。DynamoDB のバイナリセット値は次のように表されます。

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

値は実際には文字列であり、文字列はバイナリデータの base64 でエンコードされた表現です。 AWS AppSync はこの文字列をバイナリ値にデコードしてから DynamoDB に送信します。 AWS AppSync は、RFC 2045 で定義されている base64 デコードスキームを使用します。base64 アルファベットにない文字は無視されます。

以下は使用例です。

```
"attributeValues" : {
   "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }
}
```

ブール型 **BOOL**

ブール値。DynamoDB のブール値は次のように表されます。

```
{ "BOOL" : true }
```

有効な値は、`true` と `false` のみです。

以下は使用例です。

```
"attributeValues" : {
```

```
   "orderComplete" : { "BOOL" : false }
}
```

リスト型 L

サポートされているその他の DynamoDB の値のリストです。DynamoDB のリスト値は次のように表されます。

```
{ "L" : [ ... ] }
```

この値は複合値です。リストには、サポートされる DynamoDB の値 (他のリストも含む) が 0 個以上入ります。このリストには、異なる型を混在させることもできます。

以下は使用例です。

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

マップ型 M

サポートされる他の DynamoDB の値の、キーと値のペアの順序付けされていない集合を表します。DynamoDB のマップ値は次のように表されます。

```
{ "M" : { ... } }
```

マップには 0 個以上のキーと値のペアが入ります。キーは文字列である必要があり、値には、サポートされている DynamoDB の任意の値 (他のマップを含む) が使用できます。このマップには、異なる型を混在させることもできます。

以下は使用例です。

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
```

```
    }
}
```

Null 型 **NULL**

null 値です。DynamoDB の Null 値は次のように表されます。

```
{ "NULL" : null }
```

以下は使用例です。

```
"attributeValues" : {
   "phoneNumbers" : { "NULL" : null }
}
```

各タイプの詳細については、DynamoDB のドキュメントを参照してください。

型システム (レスポンスマッピング)

DynamoDB からレスポンスを受信すると、 AWS AppSync はそれを GraphQL プリミティブ型と JSON プリミティブ型に自動的に変換します。DynamoDB の各属性はデコードされ、レスポンスマッピングコンテキストで返されます。

たとえば、DynamoDB が以下を返したとします。

```
{
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

次に、 AWS AppSync DynamoDB リゾルバーはそれを次のように GraphQL および JSON 型に変換します。

```
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

このセクションでは、 AWS AppSync が次の DynamoDB スカラー、ドキュメント、およびセットタイプを変換する方法について説明します。

文字列型 **S**

単一の文字列値です。DynamoDB 文字列値が文字列として返されます。

たとえば、DynamoDB が次の DynamoDB の文字列値を返したとします。

```
{ "S" : "some string" }
```

AWS AppSync はこれを文字列に変換します。

```
"some string"
```

文字列セット型 **SS**

1 組の文字列値です。DynamoDB 文字列セット値が文字列のリストとして返されます。

たとえば、DynamoDB が次の DynamoDB 文字列セット値を返したとします。

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync はこれを文字列のリストに変換します。

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

数値型 **N**

単一の数値です。DynamoDB 数値が数字として返されます。

たとえば、DynamoDB が次の DynamoDB の数値を返したとします。

```
{ "N" : 1234 }
```

AWS AppSync はこれを数値に変換します。

```
1234
```

数値セット型 **NS**

1 組の数値です。DynamoDB 数値セットが数字のリストとして返されます。

たとえば、DynamoDB が次の DynamoDB の数値セット値を返したとします。

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync はこれを数値のリストに変換します。

```
[ 67.8, 12.2, 70 ]
```

バイナリ型 **B**

バイナリ値です。DynamoDB のバイナリ値は、その値を Base64 で表した文字列として返されます。

たとえば、DynamoDB が次の DynamoDB のバイナリ値を返したとします。

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync は、値を base64 で表現した文字列に変換します。

```
"SGVsbG8sIFdvcmxkIQo="
```

バイナリデータは、RFC 4648 と RFC 2045 で指定されているようにして Base64 エンコーディングスキームにエンコードされます。

バイナリセット型 **BS**

1 組のバイナリ値です。DynamoDB のバイナリセット値は、その値を Base64 で表した文字列のリストとして返されます。

たとえば、DynamoDB が次の DynamoDB のバイナリ値を返したとします。

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync は、値を base64 で表現した文字列のリストに変換します。

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

バイナリデータは、RFC 4648 と RFC 2045 で指定されているようにして Base64 エンコーディングスキームにエンコードされます。

ブール型 **BOOL**

ブール値。DynamoDB ブール値がブールとして返されます。

たとえば、DynamoDB が次の DynamoDB のブール値を返したとします。

```
{ "BOOL" : true }
```

AWS AppSync はこれをブール値に変換します。

```
true
```

リスト型 **L**

サポートされているその他の DynamoDB の値のリストです。DynamoDB のリスト値は値のリストとして返され、内部のそれぞれの値も変換されます。

たとえば、DynamoDB が次の DynamoDB の文字列値を返したとします。

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

AWS AppSync は、変換された値のリストに変換します。

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

マップ型 **M**

サポートされるその他の DynamoDB の値のキー/値の集合です。DynamoDB のマップ値は JSON オブジェクトとして返されます。それぞれのキー/値も変換されます。

たとえば、DynamoDB が次の DynamoDB のバイナリ値を返したとします。

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

AWS AppSync はこれを JSON オブジェクトに変換します。

```
{
   "someString" : "A string value",
   "someNumber" : 1,
   "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

Null 型 **NULL**

null 値です。

たとえば、DynamoDB が次の DynamoDB のブール値を返したとします。

```
{ "NULL" : null }
```

AWS AppSync はこれを null に変換します。

```
null
```

フィルター

Query 処理と Scan 処理を使用して DynamoDB のオブジェクトにクエリを実行する場合、オプションで、結果を評価する filter を指定して、必要な値のみを返すことができます。

Query または Scan マッピングドキュメントのフィルタマッピングセクションは以下の構造を持ちます。

```
"filter" : {
```

```
    "expression" : "filter expression"
    "expressionNames" : {
        "#name" : "name",
    },
    "expressionValues" : {
        ":value" : ... typed value
    },
}
```

各フィールドの定義は以下のようになります。

expression

クエリ式です。フィルタ式の記述方法の詳細については、「DynamoDB QueryFilter」および「DynamoDB ScanFilter」の各ドキュメントを参照してください。このフィールドの指定は必須です。

expressionNames

式の属性名のプレースホルダーを示します。キー - 値のペアの形式になります。キーは、`expression` で使用される名前のプレースホルダーに対応します。値は、DynamoDB の項目の属性名に対応する文字列である必要があります。このフィールドはオプションであり、`expression` で使用される式の属性名のプレースホルダーのみを入力します。

expressionValues

式の属性値のプレースホルダーを示します。キー - 値のペアの形式になります。キーは `expression` で使用される値のプレースホルダーに対応し、値は型付き値でなければなりません。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この指定は必須です。このフィールドはオプションであり、`expression` で使用される式の属性値のプレースホルダーのみを入力します。

例

次の例は、マッピングテンプレートのフィルターセクションです。ここでは、DynamoDB から取得されたエントリのうち、タイトルが `title` 引数で始まるもののみが返されます。

```
"filter" : {
    "expression" : "begins_with(#title, :title)",
    "expressionNames" : {
        "#title" : "title"
    },
```

```
    "expressionValues" : {
        ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
    }
}
```

条件式

PutItem、UpdateItem、および DeleteItem の各 DynamoDB 処理を使用して DynamoDB のオブジェクトをミューテーションする場合、オプションで、処理を実行する前に、DynamoDB にある既存のオブジェクトの状態に基づいてリクエストが成功するかどうかを制御する条件式を指定することができます。

AWS AppSync DynamoDB リゾルバーでは、PutItem、、UpdateItemおよび DeleteItem リクエストマッピングドキュメントで条件式を指定でき、条件が失敗してオブジェクトが更新されなかった場合に従う戦略も許可されます。

例 1

以下の PutItem マッピングドキュメントには条件式がありません。その結果、同じキーに対応する項目がすでにある場合でも、項目は DynamoDB に挿入され、それにより既存の項目が上書きされます。

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   }
}
```

例 2

次の PutItem マッピングドキュメントには条件式があります。この場合、同じキーの項目が DynamoDB に存在しない場合のみ処理が成功します。

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
```

```
    },
    "condition" : {
       "expression" : "attribute_not_exists(id)"
    }
}
```

デフォルトでは、条件チェックが失敗すると、 AWS AppSync DynamoDB リゾルバーはミューテーションのエラーを返します。ただし、 AWS AppSync DynamoDB リゾルバーには、開発者が一般的なエッジケースを処理するのに役立ついくつかの追加機能があります。

- AWS AppSync DynamoDB リゾルバーは、DynamoDB の現在の値が目的の結果と一致すると判断できる場合、オペレーションは成功したかのように扱われます。
- エラーを返す代わりに、カスタム Lambda 関数を呼び出して AWS AppSync DynamoDB リゾルバーが障害を処理する方法を決定するようにリゾルバーを設定できます。

これらの詳細については、「条件チェックでの エラーを処理する」セクションを参照してください。

DynamoDB の条件式の詳細については、「DynamoDB ConditionExpressions のドキュメント」を参照してください。

条件を指定する

`PutItem`、`UpdateItem`、および `DeleteItem` の各リクエストマッピングドキュメントはすべて、オプションで `condition` セクションが指定できます。省略した場合、条件チェックは実行されません。指定した場合、処理が成功するには、条件が true となる必要があります。

`condition` セクションは以下の構造を持ちます。

```
"condition" : {
    "expression" : "someExpression"
    "expressionNames" : {
        "#foo" : "foo"
    },
    "expressionValues" : {
        ":bar" : ... typed value
    },
    "equalsIgnore" : [ "version" ],
    "consistentRead" : true,
    "conditionalCheckFailedHandler" : {
        "strategy" : "Custom",
        "lambdaArn" : "arn:..."
```

```
    }
}
```

以下のフィールドに条件を指定します。

expression

更新式そのものを指定します。条件式の記述方法の詳細については、DynamoDB ConditionExpressions のドキュメントを参照してください。このフィールドの指定は必須です。

expressionNames

式の属性名のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される名前のプレースホルダーに対応し、値は DynamoDB の項目の属性名と一致する文字列でなければなりません。このフィールドはオプションであり、expression で使用される式の属性名のプレースホルダーのみを入力します。

expressionValues

式の属性値のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される値のプレースホルダーに対応し、値は型付き値でなければなりません。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この指定は必須です。このフィールドはオプションであり、expression で使用される式の属性値のプレースホルダーのみを入力します。

残りのフィールドは、 AWS AppSync DynamoDB リゾルバーに条件チェックの失敗を処理する方法を指示します。

equalsIgnore

`PutItem` オペレーションの使用時に条件チェックが失敗すると、 AWS AppSync DynamoDB リゾルバーは、DynamoDB の現在ある項目と、書き込みを試みた項目を比較します。これらが同じ場合、処理は成功として扱われます。`equalsIgnore` フィールドを使用して、 AWS AppSync がその比較を実行するときに無視する属性のリストを指定できます。たとえば、唯一の違いが `version` 属性である場合、オペレーションは成功として扱われます。このフィールドはオプションです。

consistentRead

条件チェックが失敗すると、 AWS AppSync は強力な整合性のある読み込みを使用して DynamoDB から項目の現在の値を取得します。このフィールドを使用して、 AWS AppSync

DynamoDB リゾルバーに、代わりに結果整合性のある読み込みを使用するように指示できます。このフィールドはオプションであり、デフォルトは `true` です。

`conditionalCheckFailedHandler`

このセクションでは、 AWS AppSync DynamoDB リゾルバーが DynamoDB の現在の値と予想される結果を比較した後に、条件チェック DynamoDB の失敗を処理する方法を指定できます。このセクションはオプションです。省略した場合、デフォルトの処理は `Reject` です。

`strategy`

AWS AppSync DynamoDB リゾルバーが DynamoDB の現在の値を期待される結果と比較した後に実行する戦略。このフィールドは必須であり、以下を値を設定できます。

`Reject`

ミューテーションは失敗し、GraphQL レスポンスにエラーが追加されます。

`Custom`

AWS AppSync DynamoDB リゾルバーは、カスタム Lambda 関数を呼び出して、条件チェックの失敗を処理する方法を決定します。`strategy` が `Custom` に設定されている場合、`lambdaArn` フィールドには、呼び出す Lambda 関数の ARN が含まれている必要があります。

`lambdaArn`

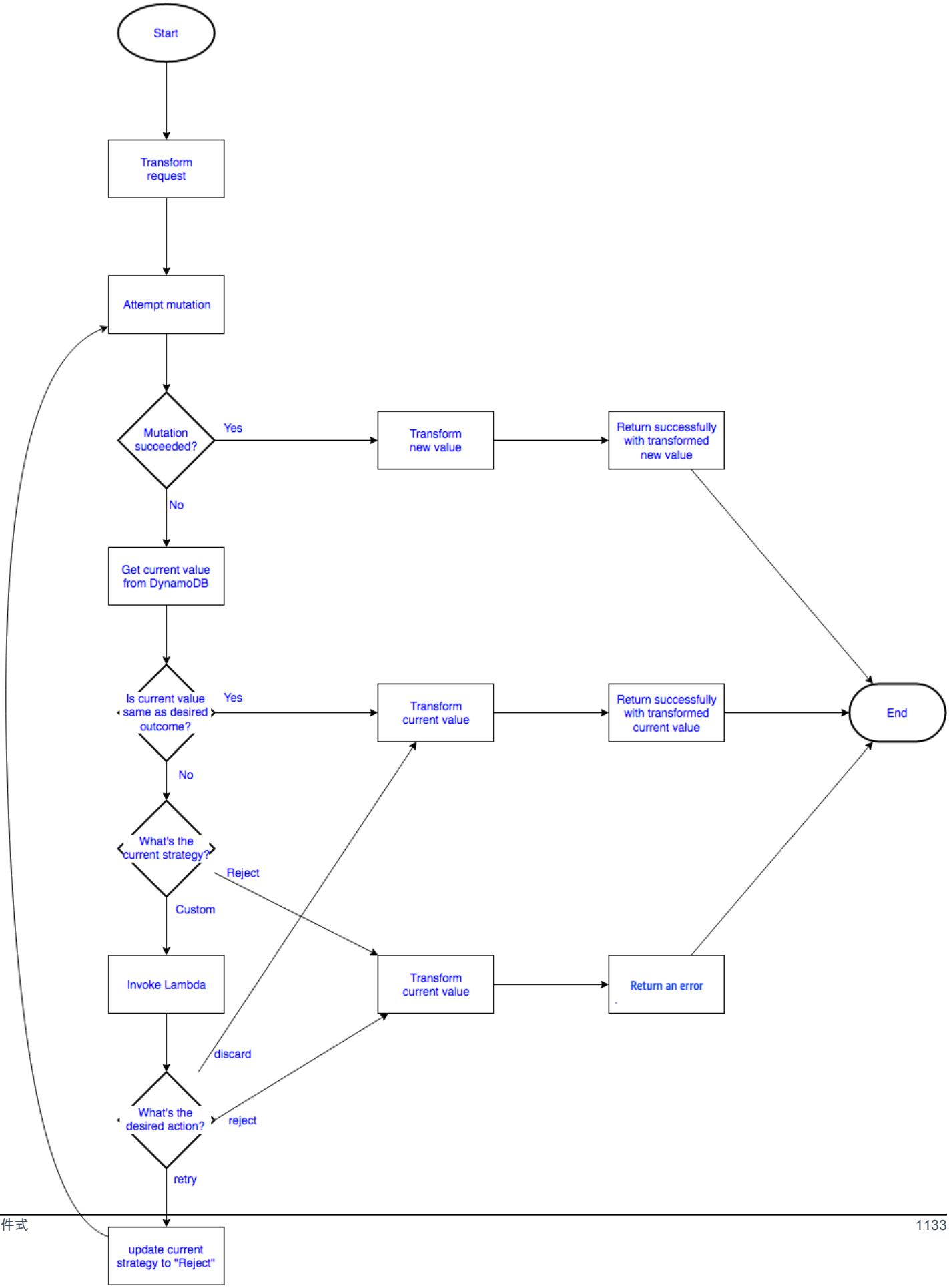
AWS AppSync DynamoDB リゾルバーが条件チェックの失敗を処理する方法を決定する、呼び出す Lambda 関数の ARN。このフィールドは、`strategy` が `Custom` に設定されている場合のみ指定する必要があります。この機能の使用方法の詳細については、「条件チェックでのエラーを処理する」を参照してください。

条件チェックでのエラーを処理する

デフォルトでは、条件チェックが失敗すると、 AWS AppSync DynamoDB リゾルバーはミューテーションのエラーと DynamoDB 内のオブジェクトの現在の値を返します。ただし、 AWS AppSync DynamoDB リゾルバーには、開発者が一般的なエッジケースを処理するのに役立ついくつかの追加機能があります。

- AWS AppSync DynamoDB リゾルバーは、DynamoDB の現在の値が目的の結果と一致すると判断できる場合、オペレーションは成功したかのように扱われます。
- エラーを返す代わりに、カスタム Lambda 関数を呼び出して AWS AppSync DynamoDB リゾルバーが障害を処理する方法を決定するようにリゾルバーを設定できます。

このプロセスのフローチャートは次のとおりです。


Start
Transform request
Attempt mutation
Mutation succeeded?
Yes
Transform new value
Return successfully with transformed new value
No
Get current value from DynamoDB
Is current value same as desired outcome?
Yes
Transform current value
Return successfully with transformed current value
End
No
What's the current strategy?
Reject
Custom
Invoke Lambda
Transform current value
Return an error
discard
What's the desired action?
reject
retry
update current strategy to "Reject"

必要な結果をチェックする

条件チェックが失敗すると、 AWS AppSync DynamoDB リゾルバーは `GetItem` DynamoDB リクエストを実行して DynamoDB から項目の現在の値を取得します。デフォルトでは、強力な整合性のある読み込みを使用しますが、`condition` ブロックの `consistentRead` フィールドを使用してこれを設定し、この値を期待した結果と比較することができます。

- `PutItem` オペレーションの場合、 AWS AppSync DynamoDB リゾルバーは、現在の値を書き込みを試みた値と比較`equalsIgnore`対象から除外します。項目が同じ場合は、処理は成功として扱われ、DynamoDB から取得された項目が返されます。それ以外の場合は、設定された処理に従います。

  たとえば、`PutItem` リクエストマッピングドキュメントが以下のようになっているとします。

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   },
   "attributeValues" : {
      "name" : { "S" : "Steve" },
      "version" : { "N" : 2 }
   },
   "condition" : {
      "expression" : "version = :expectedVersion",
      "expressionValues" : {
          ":expectedVersion" : { "N" : 1 }
      },
      "equalsIgnore": [ "version" ]
   }
}
```

  現在、DynamoDB にある項目は以下のようになりました。

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

AWS AppSync DynamoDB リゾルバーは、書き込もうとした項目を現在の値と比較し、唯一の違いは versionフィールドであることを確認できますが、 versionフィールドを無視するように設定されているため、オペレーションは成功として扱われ、DynamoDB から取得した項目を返します。

- DeleteItem オペレーションの場合、 AWS AppSync DynamoDB リゾルバーは項目が DynamoDB から返されたことを確認します。項目が返されなかった場合、処理は成功として扱われます。それ以外の場合は、設定された処理に従います。
- UpdateItem オペレーションの場合、 AWS AppSync DynamoDB リゾルバーには、現在 DynamoDB にある項目が期待される結果と一致するかどうかを判断するのに十分な情報がないため、設定された戦略に従います。

DynamoDB のオブジェクトの現在の状態が予想される結果と異なる場合、 AWS AppSync DynamoDB リゾルバーは設定された戦略に従ってミューテーションを拒否するか、Lambda 関数を呼び出して次に何をするかを決定します。

「reject」の戦略に従う

Reject 戦略に従うと、 AWS AppSync DynamoDB リゾルバーはミューテーションのエラーを返します。

たとえば、次のミューテーションリクエストが指定されたとします。

```
mutation {
    updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
        Name
        theVersion
    }
}
```

DynamoDB から返された項目が以下のようになっているとします。

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

また、レスポンスマッピングテンプレートは以下のようになっているとします。

```
{
   "id" : $util.toJson($context.result.id),
   "Name" : $util.toJson($context.result.name),
   "theVersion" : $util.toJson($context.result.version)
}
```

GraphQL レスポンスは以下のようになります。

```
{
  "data": null,
  "errors": [
    {
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      ...
    }
  ]
}
```

また、返されたオブジェクトのフィールドすべてが他のリゾルバーによって入力され、そのミューテーションが成功した場合、オブジェクトが `error` セクションに返されたときに、それらのフィールドは解決されません。

「custom」戦略に従う

`Custom` 戦略に従うと、 AWS AppSync DynamoDB リゾルバーは Lambda 関数を呼び出して、次に何をするかを決定します。Lambda 関数は以下のオプションのいずれかを選択します。

- ミューテーション を `reject` する これにより、 AWS AppSync DynamoDB リゾルバーは、前のセクションで説明したように、設定された戦略が であるかのように動作し`Reject`、ミューテーションのエラーと DynamoDB 内のオブジェクトの現在の値を返します。
- ミューテーション を `discard` する これにより、 AWS AppSync DynamoDB リゾルバーは条件チェックの失敗をサイレントに無視し、DynamoDB の値を返します。
- ミューテーション を `retry` する これにより、 AWS AppSync DynamoDB リゾルバーは新しいリクエストマッピングドキュメントでミューテーションを再試行するように指示されます。

Lambda 呼び出しリクエスト

AWS AppSync DynamoDB リゾルバーは、 で指定された Lambda 関数を呼び出しますlambdaArn。また、データソースに設定されたものと同じ service-role-arn を使用します。呼び出しのペイロードは以下の構造を持ちます。

```
{
    "arguments": { ... },
    "requestMapping": {... },
    "currentValue": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

各フィールドの定義は以下のようになります。

arguments

GraphQL ミューテーションの引数です。これは、$context.arguments のリクエストマッピングドキュメントで使用できる引数と同じです。

requestMapping

この処理のリクエストマッピングドキュメントです。

currentValue

DynamoDB のオブジェクトの現在値。

resolver

AWS AppSync リゾルバーに関する情報。

identity

呼び出し元に関する情報。これは、$context.identity のリクエストマッピングドキュメントで使用できる識別情報と同じです。

完全なペイロードの例を次に示します。

```
{
    "arguments": {
        "id": "1",
        "name": "Steve",
        "expectedVersion": 1
    },
```

```
    "requestMapping": {
        "version" : "2017-02-28",
        "operation" : "PutItem",
        "key" : {
           "id" : { "S" : "1" }
        },
        "attributeValues" : {
           "name" : { "S" : "Steve" },
           "version" : { "N" : 2 }
        },
        "condition" : {
           "expression" : "version = :expectedVersion",
           "expressionValues" : {
               ":expectedVersion" : { "N" : 1 }
           },
           "equalsIgnore": [ "version" ]
        }
    },
    "currentValue": {
        "id" : { "S" : "1" },
        "name" : { "S" : "Steve" },
        "version" : { "N" : 8 }
    },
    "resolver": {
        "tableName": "People",
        "awsRegion": "us-west-2",
        "parentType": "Mutation",
        "field": "updatePerson",
        "outputType": "Person"
    },
    "identity": {
        "accountId": "123456789012",
        "sourceIp": "x.x.x.x",
        "user": "AIDAAAAAAAAAAAAAAAAA",
        "userArn": "arn:aws:iam::123456789012:user/appsync"
    }
}
```

Lambda 呼び出しレスポンス

Lambda 関数は、呼び出しペイロードを確認し、任意のビジネスロジックを適用して、 AWS の AppSync DynamoDB リゾルバーがエラーを処理する方法を決定することができます。条件チェックで検出されたエラーを処理するために、以下の 3 つのオプションが指定できます。

- ミューテーション を `reject` する このオプションのレスポンスペイロードは次の構造を持ちます。

```
{
    "action": "reject"
}
```

これにより、 AWS AppSync DynamoDB リゾルバーは、上記のセクションで説明されているように、設定された戦略が であるかのように動作し`Reject`、ミューテーションのエラーと DynamoDB 内のオブジェクトの現在の値を返します。

- ミューテーション を `discard` する このオプションのレスポンスペイロードは次の構造を持ちます。

```
{
    "action": "discard"
}
```

これにより、 AWS AppSync DynamoDB リゾルバーは条件チェックの失敗を黙って無視し、DynamoDB の値を返します。

- ミューテーション を `retry` する このオプションのレスポンスペイロードは次の構造を持ちます。

```
{
    "action": "retry",
    "retryMapping": { ... }
}
```

これにより、 AWS AppSync DynamoDB リゾルバーは新しいリクエストマッピングドキュメントでミューテーションを再試行するように指示されます。`retryMapping` セクションの構造は DynamoDB の処理によって異なり、その処理の完全なリクエストマッピングドキュメントのサブセットとなります。

`PutItem` の場合、`retryMapping` セクションは次の構造を持ちます。`attributeValues` フィールドについては、「PutItem」を参照してください。

```
{
    "attributeValues": { ... },
    "condition": {
```

```
        "equalsIgnore" = [ ... ],
        "consistentRead" = true
    }
}
```

UpdateItem の場合、retryMapping セクションは次の構造を持ちます。update セクションについては、「UpdateItem」を参照してください。

```
{
    "update" : {
        "expression" : "someExpression"
        "expressionNames" : {
            "#foo" : "foo"
        },
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "condition": {
        "consistentRead" = true
    }
}
```

DeleteItem の場合、retryMapping セクションは次の構造を持ちます。

```
{
    "condition": {
        "consistentRead" = true
    }
}
```

使用する別の処理やキーを指定する方法はありません。 AWS AppSync DynamoDB リゾルバーは、同じオブジェクトに対して同じオペレーションの再試行のみを許可します。また、condition セクションでは conditionalCheckFailedHandler は指定できません。再試行が失敗した場合、 AWS AppSync DynamoDB リゾルバーは Reject戦略に従います。

以下は、失敗した PutItem リクエストを処理する Lambda 関数の例です。ビジネスロジックは呼び出し元を調べます。呼び出し元が jeffTheAdmin の場合は、リクエストを再試行して、現在 DynamoDB にある項目の version と expectedVersion を更新します。それ以外の場合は、ミューテーションを拒否します。

```
exports.handler = (event, context, callback) => {
    console.log("Event: "+ JSON.stringify(event));

    // Business logic goes here.

    var response;
    if ( event.identity.user == "jeffTheAdmin" ) {
        response = {
            "action" : "retry",
            "retryMapping" : {
                "attributeValues" : event.requestMapping.attributeValues,
                "condition" : {
                    "expression" : event.requestMapping.condition.expression,
                    "expressionValues" :
 event.requestMapping.condition.expressionValues
                }
            }
        }
        response.retryMapping.attributeValues.version = { "N" :
 event.currentValue.version.N + 1 }
        response.retryMapping.condition.expressionValues[':expectedVersion'] =
 event.currentValue.version

    } else {
        response = { "action" : "reject" }
    }

    console.log("Response: "+ JSON.stringify(response))
    callback(null, response)
};
```

トランザクション条件式

トランザクション条件式は、TransactWriteItems の 4 つのタイプのオペレーション (PutItem、DeleteItem、UpdateItem、ConditionCheck) すべてに対応するリクエストマッピングテンプレートで使用できます。

PutItem、DeleteItem、および UpdateItem の場合、トランザクション条件式はオプションです。ConditionCheck の場合、トランザクション条件式が必要です。

例 1

次のトランザクション DeleteItem マッピングドキュメントには条件式がありません。その結果、DynamoDB の項目が削除されます。

```
{
   "version": "2018-05-29",
   "operation": "TransactWriteItems",
   "transactItems": [
      {
         "table": "posts",
         "operation": "DeleteItem",
         "key": {
            "id": { "S" : "1" }
         }
      }
   ]
}
```

例 2

次のトランザクション DeleteItem マッピングドキュメントには、その投稿の作成者が特定の名前に等しい場合にのみオペレーションが成功できるトランザクション条件式があります。

```
{
   "version": "2018-05-29",
   "operation": "TransactWriteItems",
   "transactItems": [
      {
         "table": "posts",
         "operation": "DeleteItem",
         "key": {
            "id": { "S" : "1" }
         }
         "condition": {
            "expression": "author = :author",
            "expressionValues": {
               ":author": { "S" : "Chunyan" }
            }
         }
      }
   ]
```

```
}
```

条件チェックが失敗すると、TransactionCanceledException が発生し、エラーの詳細が $ctx.result.cancellationReasons で返されます。デフォルトでは、その条件チェックの失敗となった DynamoDB の古い項目は $ctx.result.cancellationReasons で返されます。

条件を指定する

PutItem、UpdateItem、および DeleteItem の各リクエストマッピングドキュメントはすべて、オプションで condition セクションが指定できます。省略した場合、条件チェックは実行されません。指定した場合、処理が成功するには、条件が true となる必要があります。ConditionCheck では、condition セクションを指定する必要があります。トランザクション全体が成功するためには、条件が true でなければなりません。

condition セクションは以下の構造を持ちます。

```
"condition": {
    "expression": "someExpression",
    "expressionNames": {
        "#foo": "foo"
    },
    "expressionValues": {
        ":bar": ... typed value
    },
    "returnValuesOnConditionCheckFailure": false
}
```

以下のフィールドに条件を指定します。

expression

更新式そのものを指定します。条件式の記述方法の詳細については、DynamoDB ConditionExpressions のドキュメントを参照してください。このフィールドの指定は必須です。

expressionNames

式の属性名のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される名前のプレースホルダーに対応し、値は DynamoDB の項目の属性名と一致する文字列でなければなりません。このフィールドはオプションであり、expression で使用される式の属性名のプレースホルダーのみを入力します。

expressionValues

式の属性値のプレースホルダーを示します。キー - 値のペアの形式になります。キーは expression で使用される値のプレースホルダーに対応し、値は型付き値でなければなりません。「型付き値」を指定する方法の詳細については、「型システム (リクエストマッピング)」を参照してください。この指定は必須です。このフィールドはオプションであり、expression で使用される式の属性値のプレースホルダーのみを入力します。

returnValuesOnConditionCheckFailure

条件チェックが失敗した場合に、DynamoDB の項目を取得し直すかどうかを指定します。取得された項目は `$ctx.result.cancellationReasons[$index].item` にあります。ここで `$index` は、条件チェックに失敗したリクエスト項目のインデックスです。この値のデフォルト値は true です。

計画

`GetItem`、`Scan`、`Query`、`BatchGetItem`、および `TransactGetItems` オペレーションを使用して DynamoDB のオブジェクトを読み取る場合、必要な属性を識別するプロジェクションをオプションで指定できます。プロジェクションは次のような構造で、フィルタ-に似ています。

```
"projection" : {
    "expression" : "projection expression"
    "expressionNames" : {
        "#name" : "name",
    }
}
```

各フィールドの定義は以下のようになります。

expression

プロジェクション式、これは文字列です。1 つの属性を取得するには、名前を指定します。複数の属性の場合、名前をカンマで区切る必要があります。プロジェクション式の記述の詳細については、「DynamoDB プロジェクション式」のドキュメントを参照してください。このフィールドは必須です。

expressionNames

式の属性名のプレースホルダーを示します。キー - 値のペアの形式になります。キーは、`expression` で使用される名前のプレースホルダーに対応します。値は、DynamoDB

の項目の属性名に対応する文字列である必要があります。このフィールドはオプションであり、expression で使用される式の属性名のプレースホルダーのみを入力します。expressionNames の詳細については、「Amazon DynamoDB のドキュメント」を参照してください。

例 1

次の例は、属性 author と id が DynamoDB から返される VTL マッピングテンプレートのプロジェクション セクションです。

```
"projection" : {
    "expression" : "#author, id",
    "expressionNames" : {
        "#author" : "author"
    }
}
```

> **Tip**
>
> GraphQL リクエストセレクションセットには $context.info.SelectionSetList を使用してアクセスできます。このフィールドを使うと、必要に応じてプロジェクション式を動的にフレーミングできます。

> **Note**
>
> Query と Scan のオペレーションでプロジェクション式を使用する場合、select の値は SPECIFIC_ATTRIBUTES でなければなりません。詳細については、「DynamoDB のドキュメント」を参照してください。

RDS の AWS AppSync リゾルバーマッピングテンプレートリファレンス

AWS AppSync RDS リゾルバーマッピングテンプレートを使用すると、デベロッパーは Amazon Aurora Serverless のデータ API に SQL クエリを送信し、これらのクエリの結果を取得できます。

リクエストマッピングテンプレート

RDS リクエストマッピングテンプレートは非常にシンプルです。

```
{
    "version": "2018-05-29",
    "statements": [],
    "variableMap": {},
    "variableTypeHintMap": {}
}
```

以下に示しているのは、解決済み RDS リクエストマッピングテンプレートの JSON スキーマ表現です。

```
{
    "definitions": {},
    "$schema": "https://json-schema.org/draft-07/schema#",
    "$id": "https://example.com/root.json",
    "type": "object",
    "title": "The Root Schema",
    "required": [
        "version",
        "statements",
        "variableMap"
    ],
    "properties": {
        "version": {
            "$id": "#/properties/version",
            "type": "string",
            "title": "The Version Schema",
            "default": "",
            "examples": [
                "2018-05-29"
            ],
            "enum": [
                "2018-05-29"
            ],
            "pattern": "^(.*)$"
        },
        "statements": {
            "$id": "#/properties/statements",
            "type": "array",
```

```
            "title": "The Statements Schema",
            "items": {
                "$id": "#/properties/statements/items",
                "type": "string",
                "title": "The Items Schema",
                "default": "",
                "examples": [
                    "SELECT * from BOOKS"
                ],
                "pattern": "^(.*)$"
            }
        },
        "variableMap": {
            "$id": "#/properties/variableMap",
            "type": "object",
            "title": "The Variablemap Schema"
        },
        "variableTypeHintMap": {
            "$id": "#/properties/variableTypeHintMap",
            "type": "object",
            "title": "The variableTypeHintMap Schema"
        }
    }
}
```

以下は、静的クエリを使用したリクエストマッピングテンプレートの例です。

```
{
    "version": "2018-05-29",
    "statements": [
        "select title, isbn13 from BOOKS where author = 'Mark Twain'"
    ]
}
```

バージョン

すべてのリクエストマッピングテンプレートに共通で、バージョンフィールド はテンプレートが使用するバージョンを定義します。バージョンフィールドは必須です。値「2018-05-29」は、Amazon RDS マッピングテンプレートでサポートされている唯一のバージョンです。

```
"version": "2018-05-29"
```

statements と VariableMap

statements 配列は、開発者が提供したクエリのプレースホルダーです。現在、リクエストマッピングテンプレートごとに最大 2 つのクエリがサポートされています。variableMap は、SQL ステートメントを短くして読みやすくするために使用できるエイリアスを含むオプションのフィールドです。例えば、以下のことが可能です。

```
{
"version": "2018-05-29",
    "statements": [
        "insert into BOOKS VALUES (:AUTHOR, :TITLE, :ISBN13)",
        "select * from BOOKS WHERE isbn13 = :ISBN13"
    ],
    "variableMap": {
        ":AUTHOR": $util.toJson($ctx.args.newBook.author),
        ":TITLE": $util.toJson($ctx.args.newBook.title),
        ":ISBN13": $util.toJson($ctx.args.newBook.isbn13)
    }
}
```

AWS AppSync は変数マップ値を使用して、Amazon Aurora Serverless Data API に送信される SqlParameterized クエリを構築します。SQL ステートメントは変数マップに指定されたパラメータを使用して実行されるため、SQL インジェクションのリスクが排除されます。

VariableTypeHintMap

variableTypeHintMap は、SQL パラメータの型ヒントを送信するために使用できるエイリアス型を含むオプションのフィールドです。これらのタイプヒントは、SQL ステートメント内での明示的なキャストを回避し、より短くします。例えば、以下のことが可能です。

```
{
    "version": "2018-05-29",
    "statements": [
        "insert into LOGINDATA VALUES (:ID, :TIME)",
        "select * from LOGINDATA WHERE id = :ID"
     ],
     "variableMap": {
        ":ID": $util.toJson($ctx.args.id),
        ":TIME": $util.toJson($ctx.args.time)
     },
     "variableTypeHintMap": {
```

```
        ":id": "UUID",
        ":time": "TIME"
    }
}
```

AWS AppSync は変数マップ値を使用して、Amazon Aurora Serverless Data API に送信されるクエリを構築します。また、`variableTypeHintMap` データを使用してタイプの情報を RDS に送信します。RDS がサポートしている `typeHints` については、こちらを参照してください。

OpenSearch の AWS AppSync リゾルバーマッピングテンプレートリファレンス OpenSearch

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

Amazon OpenSearch Service の AWS AppSync リゾルバーを使用すると、GraphQL を使用して、アカウントの既存の OpenSearch Service ドメインにデータを保存および取得できます。このリゾルバーにより、受信した GraphQL リクエストを OpenSearch Service リクエストにマッピングし、その後の OpenSearch Service のレスポンスを GraphQL にマッピングすることができます。このセクションでは、サポートされる OpenSearch Service オペレーションのマッピングテンプレートについて説明します。

リクエストマッピングテンプレート

ほとんどの OpenSearch Service リクエストのマッピングテンプレートは一部の違いはあるものの共通の構造になっています。以下の例では、OpenSearch Service ドメインに対して検索を実行します。このドメインでは、`post` というインデックスの下にドキュメントが整理されています。検索パラメータは `body` セクションで定義され、`query` フィールドで定義されている多くの一般的なクエリの句を使用します。この例では `"Nadia"`、または `"Bailey"`、あるいはその両方を、ドキュメントの `author` フィールドに含むドキュメントを検索します。

```
{
    "version":"2017-02-28",
    "operation":"GET",
```

```
    "path":"/post/_search",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "from":0,
            "size":50,
            "query" : {
                "bool" : {
                    "should" : [
                        {"match" : { "author" : "Nadia" }},
                        {"match" : { "author" : "Bailey" }}
                    ]
                }
            }
        }
    }
}
```

レスポンスマッピングテンプレート

他のデータソースと同様に、OpenSearch Service は GraphQL に変換する必要があるレスポンスを AWS AppSync に送信します。

ほとんどの GraphQL クエリは OpenSearch Service レスポンスから `_source` フィールドを探しています。個別のドキュメントまたはドキュメントのリストを返す検索を行うことができるので、OpenSearch Service で使用される、2 つの共通レスポンスマッピングテンプレートがあります。

結果のリスト

```
[
    #foreach($entry in $context.result.hits.hits)
      #if( $velocityCount > 1 ) , #end
        $utils.toJson($entry.get("_source"))
    #end
]
```

個別項目

```
$utils.toJson($context.result.get("_source"))
```

operation フィールド

> **Note**
> これは、リクエストマッピングテンプレートにのみ適用されます。

AWS AppSync が OpenSearch Service ドメインに送信する HTTP メソッドまたは動詞 (GET、POST、PUT、HEAD、DELETE)。キーと値の両方が文字列である必要があります。

```
"operation" : "PUT"
```

path フィールド

> **Note**
> これは、リクエストマッピングテンプレートにのみ適用されます。

AWS AppSync からの OpenSearch Service リクエストの検索パス。これはオペレーションの HTTP 動作に対する URL を作成します。キーと値の両方が文字列である必要があります。

```
"path" : "/<indexname>/_doc/<_id>"
"path" : "/<indexname>/_doc"
"path" : "/<indexname>/_search"
"path" : "/<indexname>/_update/<_id>
```

マッピングテンプレートが評価されると、このパスは、OpenSearch Service ドメインが含まれる HTTP リクエストの一部として送信されます。たとえば、前の例では次のように変換される可能性があります。

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

params フィールド

> **Note**
> これは、リクエストマッピングテンプレートにのみ適用されます。

検索実行時のアクションを指定するために使用され、一般に、query 値を body 内に設定します。ただし、レスポンスのフォーマットなど、他のいくつかの機能を設定できます。

- ヘッダ

  ヘッダー情報は、キーと値のペアです。キーと値の両方が文字列である必要があります。以下に例を示します。

  ```
  "headers" : {
      "Content-Type" : "application/json"
  }
  ```

  > **Note**
  >
  > AWS AppSync は現在、 として JSON のみをサポートしていますContent-Type。

- queryString

  一般的なオプション (JSON レスポンスのコードフォーマットなど) を指定するキーと値のペア。キーと値の両方が文字列である必要があります。たとえば、整形表示の JSON を取得する場合は、次を使用します。

  ```
  "queryString" : {
      "pretty" : "true"
  }
  ```

- body

  これはリクエストの主要部分であり、 AWS AppSync が OpenSearch Service ドメインに対して適切な形式の検索リクエストを作成できるようにします。キーはオブジェクトで構成される文字列である必要があります。2 つのデモを以下に示します。

例 1

都市が「seattle」に一致するすべてのドキュメントを返します。

```
"body":{
    "from":0,
    "size":50,
    "query" : {
```

```
        "match" : {
            "city" : "seattle"
        }
    }
}
```

例 2

都市または州が「washington」に一致するすべてのドキュメントを返します。

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "multi_match" : {
            "query" : "washington",
            "fields" : ["city", "state"]
        }
    }
}
```

渡す変数

> **Note**
>
> これは、リクエストマッピングテンプレートにのみ適用されます。

VTL ステートメントの評価の一部として変数を渡すこともできます。たとえば、次のような GraphQL クエリがあるとします。

```
query {
    searchForState(state: "washington"){
        ...
    }
}
```

マッピングテンプレートは引数として state をとることができます。

```
"body":{
```

```
    "from":0,
    "size":50,
    "query" : {
        "multi_match" : {
            "query" : "$context.arguments.state",
            "fields" : ["city", "state"]
        }
    }
}
```

VTL に含めることができるユーティリティのリストについては、「Access Request Headers」を参照してください。

Lambda の AWS AppSync リゾルバーマッピングテンプレートリファレンス

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync 関数とリゾルバーを使用して、 アカウントにある Lambda 関数を呼び出すことができます。クライアントに返す前に、リクエストペイロードと Lambda 関数からのレスポンスを調整することができます。マッピングテンプレートを使用して、呼び出されるオペレーションの特性に関して、 AWS AppSync にヒントを渡すこともできます。このセクションでは、サポートされる Lambda 操作用の異なるマッピングテンプレートについて説明します。

リクエストマッピングテンプレート

Lambda リクエストマッピングテンプレートは、Lambda 関数に関連するフィールドを処理します。

```
{
  "version": string,
  "operation": Invoke|BatchInvoke,
  "payload": any type,
  "invocationType": RequestResponse|Event
}
```

これは、解決時の Lambda リクエストマッピングテンプレートの JSON スキーマ表現です。

```
{
  "definitions": {},
  "$schema": "https://json-schema.org/draft-06/schema#",
  "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
  "type": "object",
  "properties": {
    "version": {
      "$id": "/properties/version",
      "type": "string",
      "enum": [
        "2018-05-29"
      ],
      "title": "The Mapping template version.",
      "default": "2018-05-29"
    },
    "operation": {
      "$id": "/properties/operation",
      "type": "string",
      "enum": [
        "Invoke",
        "BatchInvoke"
      ],
      "title": "The Mapping template operation.",
      "description": "What operation to execute.",
      "default": "Invoke"
    },
    "payload": {},
    "invocationType": {
      "$id": "/properties/invocationType",
      "type": "string",
      "enum": [
        "RequestResponse",
        "Event"
      ],
      "title": "The Mapping template invocation type.",
      "description": "What invocation type to execute.",
      "default": "RequestResponse"
    }
  },
  "required": [
    "version",
    "operation"
```

```
  ],
  "additionalProperties": false
}
```

以下は、ペイロードデータが GraphQL スキーマの `getPost` フィールドであり、コンテキストの引数を含む `invoke` オペレーションを使用する例です。

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $util.toJson($context.arguments)
  }
}
```

マッピングドキュメント全体は、使用する Lambda 関数に入力として渡されます。これにより、先ほどのサンプルは以下のようになります。

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "id": "postId1"
    }
  }
}
```

バージョン

すべてのリクエストマッピングテンプレートに共通で、`version` はテンプレートが使用するバージョンを定義します。`version` は必須で、静的値です。

```
"version": "2018-05-29"
```

Operation

Lambda データソースでは、`operation` フィールドで 2 つのオペレーションである `Invoke` と `BatchInvoke` を定義できます。`Invoke` オペレーションは、GraphQL フィールドリゾル

バーごとに Lambda 関数を呼び出すことを AWS AppSync に知らせます。 BatchInvoke は、現在の GraphQL フィールドのリクエストをバッチ処理するように AWS AppSync に指示します。operation フィールドは必須です。

Invoke では、解決されたリクエストマッピングテンプレートは Lambda 関数の入力ペイロードと一致します。上記の例を変更してみましょう。

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
    "payload": {
      "arguments": $util.toJson($context.arguments)
    }
}
```

これは解決され、Lambda 関数に渡されます。この関数は次のようになります。

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
    "payload": {
      "arguments": {
        "id": "postId1"
      }
    }
}
```

BatchInvoke では、バッチ内のすべてのフィールドリゾルバーにマッピングテンプレートが適用されます。簡潔にするために、 AWS AppSync は解決されたすべてのマッピングテンプレートpayload値を、マッピングテンプレートに一致する 1 つのオブジェクトのリストにマージします。次のサンプルテンプレートは、マージを示します。

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": $util.toJson($context)
}
```

このテンプレートは、次のマッピングドキュメントに解決されます。

```
{
```

```
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

payload リストの各要素は、1 回のバッチ項目に対応しています。また、Lambda 関数はリスト形式のレスポンスを返すことが予期され、次のように、各項目はリクエストで送信された順序になります。

```
[
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
 1
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
 2
  { "data": {...}, "errorMessage": null, "errorType": null }  // result for batch item
 3
]
```

ペイロード

payload フィールドは、任意の正しい形式の JSON を Lambda 関数に渡す際に使用するコンテナです。operation フィールドが に設定されている場合BatchInvoke、 AWS AppSync は既存のpayload値をリストにラップします。payload フィールドはオプションです。

呼び出しタイプ

Lambda データソースでは、2 つの呼び出しタイプ RequestResponse と Event を定義できます。呼び出しタイプは、Lambda API で定義されている呼び出しタイプと同義です。RequestResponse 呼び出しタイプ lets AWS AppSync は Lambda 関数を同期的に呼び出してレスポンスを待ちます。Event 呼び出しにより、Lambda 関数を非同期的に呼び出すことができます。Lambda が Event 呼び出しタイプのリクエストを処理する方法の詳細については、「非同期呼び出し」を参照してください。invocationType フィールドはオプションです。このフィールドがリクエストに含まれていない場合、 AWS AppSync はデフォルトで RequestResponse 呼び出しタイプになります。

invocationType フィールドでは、解決されたリクエストは Lambda 関数の入力ペイロードと一致します。上記の例を変更してみましょう。

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event"
  "payload": {
    "arguments": $util.toJson($context.arguments)
  }
}
```

これは解決され、Lambda 関数に渡されます。この関数は次のようになります。

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

BatchInvoke オペレーションをEvent呼び出しタイプフィールドと組み合わせて使用すると、AWS AppSync は上記の同じ方法でフィールドリゾルバーをマージし、リクエストは値payloadのリストである を使用して非同期イベントとして Lambda 関数に渡されます。Event 呼び出しタイプのリゾルバーのリゾルバーキャッシュは、キャッシュヒットがあった場合に Lambda に送信されないため、無効にすることをお勧めします。

レスポンスマッピングテンプレート

他のデータソースと同様に、Lambda 関数は GraphQL タイプに変換する必要があるレスポンスを AWS AppSync に送信します。

Lambda 関数の結果は、Velocity Template Language (VTL) $context.result プロパティを通じて使用できる context オブジェクトで設定されます。

Lambda 関数のレスポンスの形状と GraphQL タイプの形状が正確に一致する場合は、以下のレスポンスマッピングテンプレートを使用して、レスポンスを転送できます。

```
$util.toJson($context.result)
```

レスポンスマッピングテンプレートに適用される形状の制限や必須フィールドはありません。ただし、GraphQL が厳密に型指定されているので、解決されたマッピングテンプレートは予想される GraphQL タイプに一致する必要があります。

Lambda 関数のバッチ処理されたレスポンス

`operation` フィールドが `BatchInvoke` に設定されている場合、 AWS AppSync は Lambda 関数から返される項目のリストを想定します。 AWS AppSync が各結果を元のリクエスト項目にマッピングするには、レスポンスリストのサイズと順序が一致している必要があります。レスポンスリストに `null` 項目がある場合も有効です。その場合、`$ctx.result` が null に設定されます。

ダイレクトLambda リゾルバー

マッピングテンプレートの使用を完全に回避する場合、 AWS AppSync は Lambda 関数にデフォルトのペイロードを提供し、GraphQL タイプにデフォルトの Lambda 関数レスポンスを提供できます。リクエストテンプレート、レスポンステンプレート、またはどちらでもない場合でも指定でき、AWS AppSync はそれに応じて処理します。

ダイレクト Lambda リクエストマッピングテンプレート

リクエストマッピングテンプレートが指定されていない場合、 AWS AppSync は `Invoke` オペレーションとして Lambda 関数に直接`Context`オブジェクトを送信します。`Context` オブジェクトの構造の詳細については、「AWS AppSync リゾルバーマッピングテンプレートコンテキストリファレンス」を参照してください。

ダイレクト Lambda レスポンスマッピングテンプレート

レスポンスマッピングテンプレートが指定されていない場合、 AWS AppSync は Lambda 関数のレスポンスを受信したときに 2 つの処理のいずれかを実行します。リクエストマッピングテンプレートを指定しなかった場合、またはバージョン `2018-05-29` でリクエストマッピングテンプレートを提供した場合、応答は次のレスポンスマッピングテンプレートと同様になります。

```
#if($ctx.error)
     $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
 #end
 $util.toJson($ctx.result)
```

バージョン `2017-02-28` のテンプレートを提供した場合、レスポンスロジックは次のレスポンスマッピングテンプレートと同様に機能します。

```
$util.toJson($ctx.result)
```

表面的には、マッピングテンプレートのバイパスは、前の例に示すように、特定のマッピングテンプレートを使用する場合と同様に機能します。ただし、裏では、マッピングテンプレートの評価は完全に回避されます。テンプレート評価ステップはバイパスされるため、一部のシナリオでは、評価が必要なレスポンスマッピングテンプレートを持つ Lambda 関数と比較して、アプリケーションのレスポンスのオーバーヘッドとレイテンシーが少なくなる場合があります。

ダイレクト Lambda リゾルバーレスポンスのカスタムエラー処理

カスタム例外を発生させることにより、ダイレクト Lambda リゾルバーによって呼び出される Lambda 関数からのエラーレスポンスをカスタマイズできます。次の例では、JavaScript を使用してカスタム例外を作成する方法を示します。

```
class CustomException extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomException";
  }
}

throw new CustomException("Custom message");
```

例外が発生すると、`errorType` と `errorMessage` が、それぞれ個別に、スローされるカスタムエラーの `name` と `message` になります。

`errorType` が の場合`UnauthorizedException`、 AWS AppSync はカスタムメッセージの代わりにデフォルトのメッセージ (`"You are not authorized to make this call."`) を返します。

以下のスニペットは、カスタム `errorType` を示す GraphQL レスポンスの例です。

```
{
  "data": {
    "query": null
  },
  "errors": [
    {
      "path": [
        "query"
      ],
```

```
      "data": null,
      "errorType": "CustomException",
      "errorInfo": null,
      "locations": [
        {
          "line": 5,
          "column": 10,
          "sourceName": null
        }
      ],
      "message": "Custom Message"
    }
  ]
}
```

ダイレクト Lambda リゾルバー: バッチ処理が有効

Direct Lambda リゾルバーのバッチ処理を有効にするには、リゾルバーで `maxBatchSize` を設定します。`maxBatchSize` をダイレクト Lambda リゾルバーで `0` より大きい値に設定すると、 AWS AppSync は Lambda 関数に最大サイズ `maxBatchSize` のリクエストをバッチで送信します。

ダイレクト Lambda リゾルバーで `maxBatchSize` を `0` に設定すると、バッチ処理がオフになります。

Lambda リゾルバーによるバッチ処理の仕組みの詳細については、「高度なユースケース: バッチ処理」を参照してください。

リクエストマッピングテンプレート

バッチ処理が有効で、リクエストマッピングテンプレートが提供されていない場合、 AWS AppSync は、`Context` オブジェクトのリストを `BatchInvoke` 操作として Lambda 関数に直接送信します。

レスポンスマッピングテンプレート

バッチ処理が有効で、レスポンスマッピングテンプレートが提供されていない場合、レスポンスロジックは次のレスポンスマッピングテンプレートと同様に機能します。

```
#if( $context.result && $context.result.errorMessage )
      $utils.error($context.result.errorMessage, $context.result.errorType,
      $context.result.data)
#else
      $utils.toJson($context.result.data)
#end
```

Lambda 関数は、送信された Context オブジェクトのリストと同じ順序で結果のリストを返す必要があります。特定の結果に対して errorMessage と errorTypeを指定することで、個々のエラーを返すことができます。リストの各結果には、次の形式があります。

```
{
   "data" : { ... }, // your data
   "errorMessage" : { ... }, // optional, if included an error entry is added to the
 "errors" object in the AppSync response
   "errorType" : { ... } // optional, the error type
}
```

> **Note**
>
> 結果オブジェクトの他のフィールドは現時点では無視されます。

Lambda からのエラーの処理

Lambda 関数に例外またはエラーを投げると、すべての結果に対してエラーを返すことができます。バッチリクエストのペイロードリクエストまたはレスポンスサイズが大きすぎる場合、Lambda はエラーを返します。その場合は、maxBatchSize を小さくするか、レスポンスペイロードのサイズを小さくすることを検討してください。

個別のエラー処理については、「個々のエラーを返す」を参照してください。

サンプル Lambda 関数

以下のスキーマを使用して、Post.relatedPosts フィールドリゾルバー用のダイレクト Lambda リゾルバーを作成し、maxBatchSize を 0 を超える値に設定することでバッチ処理を有効にできます。

```
schema {
    query: Query
    mutation: Mutation
}

type Query {
    getPost(id:ID!): Post
    allPosts: [Post]
}
```

```
type Mutation {
    addPost(id: ID!, author: String!, title: String, content: String, url: String):
 Post!
}

type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    relatedPosts: [Post]
}
```

次のクエリでは、relatedPosts を解決するリクエストのバッチとともに Lambda 関数が呼び出されます。

```
query getAllPosts {
  allPosts {
    id
    relatedPosts {
      id
    }
  }
}
```

Lambda 関数の簡単な実装を以下に示します。

```
const posts = {
  1: {
    id: '1',
    title: 'First book',
    author: 'Author1',
    url: 'https://amazon.com/',
    content:
      'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
 AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1',
    ups: '100',
    downs: '10',
  },
  2: {
```

```
    id: '2',
    title: 'Second book',
    author: 'Author2',
    url: 'https://amazon.com',
    content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT',
    ups: '100',
    downs: '10',
  },
  3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null, ups:
 null, downs: null },
  4: {
    id: '4',
    title: 'Fourth book',
    author: 'Author4',
    url: 'https://www.amazon.com/',
    content:
      'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4',
    ups: '1000',
    downs: '0',
  },
  5: {
    id: '5',
    title: 'Fifth book',
    author: 'Author5',
    url: 'https://www.amazon.com/',
    content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE
 TEXT AUTHOR 5 SAMPLE TEXT',
    ups: '50',
    downs: '0',
  },
}

const relatedPosts = {
  1: [posts['4']],
  2: [posts['3'], posts['5']],
  3: [posts['2'], posts['1']],
  4: [posts['2'], posts['1']],
  5: [],
}
exports.handler = async (event) => {
  console.log('event ->', event)
  // retrieve the ID of each post
```

```
  const ids = event.map((context) => context.source.id)
  // fetch the related posts for each post id
  const related = ids.map((id) => relatedPosts[id])

  // return the related posts; or an error if none were found
  return related.map((r) => {
    if (r.length > 0) {
      return { data: r }
    } else {
      return { data: null, errorMessage: 'Not found', errorType: 'ERROR' }
    }
  })
}
```

EventBridge のAWS AppSync リゾルバーマッピングテンプレートリファレンス

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

EventBridge データソースで使用される AWS AppSync リゾルバーマッピングテンプレートを使用すると、Amazon EventBridge バスにカスタムイベントを送信できます。

リクエストマッピングテンプレート

`PutEvents` リクエストマッピングテンプレートを使用すると、複数のカスタムイベントを EventBridge イベントバスに送信できます。 マッピングドキュメントの構造は次のとおりです。

```
{
    "version" : "2018-05-29",
    "operation" : "PutEvents",
    "events" : [{}]
}
```

次の例は、EventBridge のリクエストマッピングテンプレートの例です。

```
{
```

```
    "version": "2018-05-29",
    "operation": "PutEvents",
    "events": [{
        "source": "com.mycompany.myapp",
        "detail": {
            "key1" : "value1",
            "key2" : "value2"
        },
        "detailType": "myDetailType1"
    },
    {
        "source": "com.mycompany.myapp",
        "detail": {
            "key3" : "value3",
            "key4" : "value4"
        },
        "detailType": "myDetailType2",
        "resources" : ["Resource1", "Resource2"],
        "time" : "2023-01-01T00:30:00.000Z"
    }

    ]
}
```

レスポンスマッピングテンプレート

PutEvents 操作が成功すると、EventBridge からの応答が $ctx.result に含まれます。

```
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end
  $util.toJson($ctx.result)
```

InternalExceptions や Timeouts などの PutEvents 操作の実行中に発生したエラーは、$ctx.error に表示されます。EventBridge の一般的なエラーのリストについては、EventBridge の一般的なエラーのリファレンスを参照してください。

result は、次の形式になります。

```
{
    "Entries" [
        {
```

```
            "ErrorCode" : String,
            "ErrorMessage" : String,
            "EventId" : String
        }
    ],
    "FailedEntryCount" : number
}
```

- エントリ

  取り込まれたイベントは、成功と失敗の両方の結果になります。取り込みが成功すると、エントリには EventID が含まれます。それ以外の場合は、ErrorCode と ErrorMessage を使用してエントリの問題を特定できます。

  各レコードの応答要素のインデックスは、リクエスト配列のインデックスと同じです。

- FailedEntryCount

  失敗したエントリの数。この値は整数として表されます。

PutEvents のレスポンスの詳細については、「PutEvents」を参照してください。

サンプルレスポンスの例: 1

次の例は、2 つのイベントが成功する PutEvents 操作です。

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ],
    "FailedEntryCount" : 0
}
```

サンプルレスポンスの例: 2

次の例は、3 つのイベント (2 つの成功、1 つの失敗) がある PutEvents 操作です。

```
{
```

```
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        },
        {
            "ErrorCode" : "SampleErrorCode",
            "ErrorMessage" : "Sample Error Message"
        }
    ],
    "FailedEntryCount" : 1
}
```

PutEvents フィールド

PutEvents には、次のマッピングテンプレートフィールドが含まれています。

- バージョン

  すべてのリクエストマッピングテンプレートに共通で、version フィールドはテンプレートが使用するバージョンを定義します。このフィールドは必須です。値 2018-05-29 は、EventBridge マッピングテンプレートでサポートされている唯一のバージョンです。

- 操作

  サポートされている操作は PutEvents のみです。この操作により、カスタムイベントをイベントバスに追加できます。

- イベント

  イベントバスに追加されるイベントの配列。この配列には 1 ~ 10 個の項目が割り当てられている必要があります。

  Event オブジェクトは、以下のフィールドを持つ有効な JSON オブジェクトです。

  - "source": イベントのソースを識別する文字列。
  - "detail": イベントに関する情報をアタッチするのに使用できる JSON オブジェクト。このフィールドは空のマップ ({ }) でもかまいません。
  - "detailType: イベントの種類を識別する文字列。

- "resources": イベントに関わるリソースを識別する文字列の JSON 配列 このフィールドは、空白の配列でもかまいません。
- "time": 文字列として提供されるイベントのタイムスタンプ。これは RFC3339 タイムスタンプ形式に従う必要があります。

以下のスニペットは有効な Event オブジェクトの例です。

例 1

```
{
    "source" : "source1",
    "detail" : {
        "key1" : [1,2,3,4],
        "key2" : "strval"
    },
    "detailType" : "sampleDetailType",
    "resources" : ["Resouce1", "Resource2"],
    "time" : "2022-01-10T05:00:10Z"
}
```

例 2

```
{
    "source" : "source1",
    "detail" : {},
    "detailType" : "sampleDetailType"
}
```

例 3

```
{
    "source" : "source1",
    "detail" : {
        "key1" : 1200
    },
    "detailType" : "sampleDetailType",
    "resources" : []
}
```

None データソースの AWS AppSync リゾルバーマッピングテンプレートリファレンス

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

None 型のデータソースで使用される AWS AppSync リゾルバーマッピングテンプレートを使用すると、 AWS AppSync ローカルオペレーションのリクエストをシェイプできます。

リクエストマッピングテンプレート

マッピングテンプレートはシンプルで、payload フィールド経由で可能な限り多くのコンテキスト情報を渡すことができます。

```
{
   "version": string,
   "payload": any type
}
```

以下に示しているのは、解決済みリクエストマッピングテンプレートの JSON スキーマ表現です。

```
{
    "definitions": {},
    "$schema": "https://json-schema.org/draft-06/schema#",
    "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
    "type": "object",
    "properties": {
        "version": {
            "$id": "/properties/version",
            "type": "string",
            "enum": [
                "2018-05-29"
            ],
            "title": "The Mapping template version.",
            "default": "2018-05-29"
        },
        "payload": {}
```

```
    },
    "required": [
        "version"
    ],
    "additionalProperties": false
}
```

以下に示しているのは、VTL コンテキストプロパティ $context.arguments 経由でフィールド引数を渡すようにした例です。

```
{
    "version": "2018-05-29",
    "payload": $util.toJson($context.arguments)
}
```

payload フィールドの値は、レスポンスのマッピングテンプレートに転送され、VTL コンテキストプロパティ ($context.result) で使用できます。

次の例では、payload フィールドの値が補間されたものを表します。

```
{
    "id": "postId1"
}
```

バージョン

すべてのリクエストマッピングテンプレートに共通で、version フィールドはテンプレートが使用するバージョンを定義します。

version フィールドは必須です。

例:

```
"version": "2018-05-29"
```

ペイロード

payload フィールドは、任意の正しい形式の JSON をレスポンスマッピングテンプレートに渡すために使用できるコンテナです。

payload フィールドはオプションです。

レスポンスマッピングテンプレート

データソースがないため、payload フィールドの値がレスポンスマッピングテンプレートに転送され、context オブジェクトに設定されます。このオブジェクトは VTL $context.result プロパティ経由で利用できます。

payload フィールド値の形状と GraphQL タイプの形状が正確に一致する場合、以下のレスポンスマッピングテンプレートを使用して、レスポンスを転送できます。

```
$util.toJson($context.result)
```

レスポンスマッピングテンプレートに適用される形状の制限や必須フィールドはありません。ただし、GraphQL が厳密に型指定されているので、解決されたマッピングテンプレートは予想される GraphQL タイプに一致する必要があります。

HTTP の AWS AppSync リゾルバーマッピングテンプレートリファレンス

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちら で APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

AWS AppSync HTTP リゾルバーマッピングテンプレートを使用すると、 AWS AppSync から任意の HTTP エンドポイントにリクエストを送信し、HTTP エンドポイントからのレスポンスを AWS AppSync に返すことができます。マッピングテンプレートを使用すると、呼び出すオペレーションの性質に関するヒントを AWS AppSync に提供できます。このセクションでは、サポートされる HTTP リゾルバーのマッピングテンプレートについて説明します。

リクエストマッピングテンプレート

```
{
    "version": "2018-05-29",
    "method": "PUT|POST|GET|DELETE|PATCH",
    "params": {
```

```
        "query": Map,
        "headers": Map,
        "body": any
    },
    "resourcePath": string
}
```

HTTP リクエストマッピングテンプレートが解決されると、リクエストマッピングテンプレートの JSON スキーマ表現は以下のようになります。

```
{
    "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
    "type": "object",
    "properties": {
        "version": {
        "$id": "/properties/version",
        "type": "string",
        "title": "The Version Schema ",
        "default": "",
        "examples": [
            "2018-05-29"
        ],
        "enum": [
            "2018-05-29"
        ]
        },
        "method": {
        "$id": "/properties/method",
        "type": "string",
        "title": "The Method Schema ",
        "default": "",
        "examples": [
            "PUT|POST|GET|DELETE|PATCH"
        ],
        "enum": [
            "PUT",
            "PATCH",
            "POST",
            "DELETE",
            "GET"
        ]
        },
        "params": {
```

```
        "$id": "/properties/params",
        "type": "object",
        "properties": {
            "query": {
            "$id": "/properties/params/properties/query",
            "type": "object"
            },
            "headers": {
            "$id": "/properties/params/properties/headers",
            "type": "object"
            },
            "body": {
            "$id": "/properties/params/properties/body",
            "type": "string",
            "title": "The Body Schema ",
            "default": "",
            "examples": [
                ""
            ]
            }
        }
        },
        "resourcePath": {
        "$id": "/properties/resourcePath",
        "type": "string",
        "title": "The Resourcepath Schema ",
        "default": "",
        "examples": [
            ""
        ]
        }
    },
    "required": [
        "version",
        "method",
        "resourcePath"
    ]
}
```

以下は、ボディを text/plain とする HTTP POST リクエストの例です。

```
{
    "version": "2018-05-29",
```

```
    "method": "POST",
    "params": {
        "headers":{
        "Content-Type":"text/plain"
        },
        "body":"this is an example of text body"
    },
    "resourcePath": "/"
}
```

バージョン

> **Note**
>
> これは、リクエストマッピングテンプレートにのみ適用されます。

テンプレートが使用するバージョンを定義します。version はすべてのリクエストマッピングテンプレートに共通であり、必須です。

```
"version": "2018-05-29"
```

[メソッド]

> **Note**
>
> これは、リクエストマッピングテンプレートにのみ適用されます。

AWS AppSync が HTTP エンドポイントに送信する HTTP メソッドまたは動詞 (GET、POST、PUT、PATCH、または DELETE)。

```
"method": "PUT"
```

ResourcePath

> **Note**
>
> これは、リクエストマッピングテンプレートにのみ適用されます。

アクセス対象のリソースパスです。リソースパスは、HTTP データソースのエンドポイントとともに、 AWS AppSync サービスがリクエストを行う URL を形成します。

```
"resourcePath": "/v1/users"
```

マッピングテンプレートが評価されると、このパスは、HTTP エンドポイントが含まれる HTTP リクエストの一部として送信されます。たとえば、前の例では次のように変換される可能性があります。

```
PUT <endpoint>/v1/users
```

Params フィールド

> **Note**
>
> これは、リクエストマッピングテンプレートにのみ適用されます。

検索実行時のアクションを指定するために使用され、一般に、query 値を body 内に設定します。ただし、レスポンスのフォーマットなど、他のいくつかの機能を設定できます。

ヘッダ

ヘッダー情報は、キーと値のペアです。キーと値の両方が文字列である必要があります。

例:

```
"headers" : {
    "Content-Type" : "application/json"
}
```

現在サポートされている Content-Type ヘッダーは以下のとおりです。

```
text/*
application/xml
application/json
application/soap+xml
application/x-amz-json-1.0
```

```
application/x-amz-json-1.1
application/vnd.api+json
application/x-ndjson
```

注意: 以下の HTTP ヘッダーを設定することはできません。

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

一般的なオプション (JSON レスポンスのコードフォーマットなど) を指定するキーと値のペア。キーと値の両方が文字列である必要があります。次の例では、`?type=json` としてクエリ文字列を送信する方法を示しています。

```
"query" : {
    "type" : "json"
}
```

body

ボディには、設定の際に選択する HTTP リクエストボディが含まれています。リクエストボディは、コンテンツタイプが charset に指定されている場合を除き、常に UTF-8 でエンコードされた文字列です。

```
"body":"body string"
```

レスポンス

例を参照してください。こちらへ。

HTTPS エンドポイント AWS AppSync で によって認識される認証機関 (CA)

> **Note**
>
> Let's Encryptは、identrustおよびisrgrootx1証明書を通じて受理されます。Let's Encrypt を使用する場合、お客様が必要なアクションはありません。

現時点では、HTTPS を使用する場合、自己署名証明書は HTTP リゾルバーではサポートされていません。 AWS AppSync は、HTTPS の SSL/TLS 証明書を解決するときに次の認証局を認識します。

の既知のルート証明書 AWS AppSync

名前	日付	SHA1 フィンガープリント
digicertassuredidrootca	2018 年 4 月 21 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
trustcenterclass2caii	2018 年 4 月 21 日	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
thawtepremiumserverca	2018 年 4 月 21 日	E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:66:66
cia-crt-g3-02-ca	2016 年 11 月 23 日	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
swisssignplatinumg2ca	2018 年 4 月 21 日	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66
swisssignsilverg2ca	2018 年 4 月 21 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
thawteserverca	2018 年 4 月 21 日	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
equifaxsecureebusinessca1	2018 年 4 月 21 日	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4

名前	日付	SHA1 フィンガープリント
securetrustca	2018 年 4 月 21 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
utnuserfirstclientauthemailca	2018 年 4 月 21 日	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
thawtepersonalfreemailca	2018 年 4 月 21 日	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
affirmtrustnetworkingca	2018 年 4 月 21 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
entrustevca	2018 年 4 月 21 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
utnuserfirsthardwareca	2018 年 4 月 21 日	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
certumca	2018 年 4 月 21 日	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
addtrustclass1ca	2018 年 4 月 21 日	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
entrustrootcag2	2018 年 4 月 21 日	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
equifaxsecureca	2018 年 4 月 21 日	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
quovadisrootca3	2018 年 4 月 21 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	2018 年 4 月 21 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
digicertglobalrootg2	2018 年 4 月 21 日	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4

名前	日付	SHA1 フィンガープリント
digicerthighassurance evrootca	2018 年 4 月 21 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
secomvalicertclass1ca	2018 年 4 月 21 日	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
equifaxsecureglobalebusinessca1	2018 年 4 月 21 日	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
geotrustuniversalca	2018 年 4 月 21 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
deprecateditsecca	2012 年 1 月 27 日	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
verisignclass3ca	2018 年 4 月 21 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
thawteprimaryrootcag3	2018 年 4 月 21 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
thawteprimaryrootcag2	2018 年 4 月 21 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
deutschetelekomrootca2	2018 年 4 月 21 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
buypassclass3ca	2018 年 4 月 21 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
utnuserfirstobjectca	2018 年 4 月 21 日	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46
geotrustprimaryca	2018 年 4 月 21 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
buypassclass2ca	2018 年 4 月 21 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99

名前	日付	SHA1 フィンガープリント
baltimorecodesigningca	2018 年 4 月 21 日	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D
verisignclass1ca	2018 年 4 月 21 日	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1
baltimorecybertrustca	2018 年 4 月 21 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
starfieldclass2ca	2018 年 4 月 21 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
camerfirmachamberscommerceca	2018 年 4 月 21 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
ttelesecglobalrootclass3ca	2018 年 4 月 21 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
verisignclass3g5ca	2018 年 4 月 21 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
ttelesecglobalrootclass2ca	2018 年 4 月 21 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
trustcenteruniversalcai	2018 年 4 月 21 日	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
verisignclass3g4ca	2018 年 4 月 21 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3g3ca	2018 年 4 月 21 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
xrampglobalca	2018 年 4 月 21 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
amzninternalrootca	2008 年 12 月 12 日	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06

名前	日付	SHA1 フィンガープリント
certplusclass3primaryca	2018 年 4 月 21 日	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
certumtrustednetworkca	2018 年 4 月 21 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
verisignclass3g2ca	2018 年 4 月 21 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
globalsignr3ca	2018 年 4 月 21 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
utndatacorpsgcca	2018 年 4 月 21 日	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
secomscrootca2	2018 年 4 月 21 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
gtecybertrustgloballca	2018 年 4 月 21 日	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
secomscrootca1	2018 年 4 月 21 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
affirmtrustcommercialca	2018 年 4 月 21 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
trustcenterclass4caii	2018 年 4 月 21 日	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
verisignuniversalrootca	2018 年 4 月 21 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
globalsignr2ca	2018 年 4 月 21 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
certplusclass2primaryca	2018 年 4 月 21 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB

名前	日付	SHA1 フィンガープリント
digicertglobalrootca	2018 年 4 月 21 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
globalsignca	2018 年 4 月 21 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
thawteprimaryrootca	2018 年 4 月 21 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
starfieldrootg2ca	2018 年 4 月 21 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
geotrustglobalca	2018 年 4 月 21 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
soneraclass2ca	2018 年 4 月 21 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
verisigntsaca	2018 年 4 月 21 日	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
soneraclass1ca	2018 年 4 月 21 日	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
quovadisrootca	2018 年 4 月 21 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
affirmtrustpremiumecccca	2018 年 4 月 21 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
starfieldservicesrootg2ca	2018 年 4 月 21 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
valicertclass2ca	2018 年 4 月 21 日	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
comodoaaaca	2018 年 4 月 21 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49

名前	日付	SHA1 フィンガープリント
aolrootca2	2018 年 4 月 21 日	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
keynectisrootca	2018 年 4 月 21 日	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
addtrustqualifiedca	2018 年 4 月 21 日	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
aolrootca1	2018 年 4 月 21 日	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignclass2g3ca	2018 年 4 月 21 日	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
addtrustexternalca	2018 年 4 月 21 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
verisignclass2g2ca	2018 年 4 月 21 日	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
geotrustprimarycag3	2018 年 4 月 21 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycag2	2018 年 4 月 21 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
swisssigngoldg2ca	2018 年 4 月 21 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
entrust2048ca	2018 年 4 月 21 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
chunghwaepkirootca	2018 年 4 月 21 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
camerfirmachambersignca	2018 年 4 月 21 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C

名前	日付	SHA1 フィンガープリント
camerfirmachambers ca	2018 年 4 月 21 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50: BA:9E:A8:7E:FE:9A:CE:3C
godaddyclass2ca	2018 年 4 月 21 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0: D7:77:70:02:8F:20:EE:E4
affirmtrustpremium ca	2018 年 4 月 21 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F: 7D:6A:06:65:26:32:28:27
verisignclass1g3ca	2018 年 4 月 21 日	20:42:85:DC:F7:EB:76:41:95:57:8E:13: 6B:D4:B7:D1:E9:8E:46:A5
secomevrootca1	2018 年 4 月 21 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8: 90:8F:FD:28:86:65:64:7D
verisignclass1g2ca	2018 年 4 月 21 日	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B: 56:16:7F:62:F5:32:E5:47
amzninternalinfose ccag3	2015 年 2 月 27 日	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6: 5E:75:32:9B:A8:78:2E:F6
cia-crt-g3-01-ca	2016 年 11 月 23 日	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95: 08:ED:46:82:39:4D:ED:E2
godaddyrootg2ca	2018 年 4 月 21 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62: A7:9F:45:C2:54:FD:E6:8B
digicertassuredidr ootca	2018 年 4 月 21 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E: 4B:DF:B5:A8:99:B2:4D:43
microseceszignoroo tca2009	2018 年 4 月 21 日	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37: 7D:54:DA:91:E1:01:31:8E
affirmtrustcommerc ial	2018 年 4 月 21 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80: DC:E9:6E:2C:C7:B2:78:B7
comodoecccertifica tionauthority	2018 年 4 月 21 日	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50: B6:56:3B:8E:2D:93:C3:11

名前	日付	SHA1 フィンガープリント
cadisigrootr2	2018 年 4 月 21 日	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
swisssignsilvercag2	2018 年 4 月 21 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
securetrustca	2018 年 4 月 21 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
cadisigrootr1	2018 年 4 月 21 日	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
accvraiz1	2018 年 4 月 21 日	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
entrustrootcertificationauthority	2018 年 4 月 21 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
camerfirmaglobalchambersignroot	2018 年 4 月 21 日	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
dstacescax6	2018 年 4 月 21 日	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
identrustpublicsectorrootca1	2018 年 4 月 21 日	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
starfieldrootcertificateauthorityg2	2018 年 4 月 21 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
secureglobalca	2018 年 4 月 21 日	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
eecertificationcentrerootca	2018 年 4 月 21 日	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
opentrustrootcag3	2018 年 4 月 21 日	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8A:A6

名前	日付	SHA1 フィンガープリント
teliasonerarootcav1	2018 年 4 月 21 日	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
autoridaddecertificacionfirmaprofesionalcifa62634068	2018 年 4 月 21 日	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
opentrustrootcag2	2018 年 4 月 21 日	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	2018 年 4 月 21 日	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:D5:7E
globalsigneccrootcar5	2018 年 4 月 21 日	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
globalsigneccrootcar4	2018 年 4 月 21 日	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
izenpecom	2018 年 4 月 21 日	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
turktrustelektroniksertifikahizmetsaglayicisih5	2018 年 4 月 21 日	C4:18:F6:4D:46:D1:DF:00:3D:27:30:13:72:43:A9:12:11:C6:75:FB
gdcatrustauthr5root	2018 年 4 月 21 日	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
dtrustrootclass3ca22009	2018 年 4 月 21 日	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
quovadisrootca3	2018 年 4 月 21 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85

名前	日付	SHA1 フィンガープリント
quovadisrootca2	2018 年 4 月 21 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
geotrustprimarycertificationauthorityg3	2018 年 4 月 21 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycertificationauthorityg2	2018 年 4 月 21 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
oistewisekeyglobalrootgbca	2018 年 4 月 21 日	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
addtrustexternalroot	2018 年 4 月 21 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
chambersofcommerceroot2008	2018 年 4 月 21 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
digicertglobalrootg3	2018 年 4 月 21 日	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E
comodoaaaservicesroot	2018 年 4 月 21 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
digicertglobalrootg2	2018 年 4 月 21 日	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
certinomisrootca	2018 年 4 月 21 日	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:88:A8
oistewisekeyglobalrootgaca	2018 年 4 月 21 日	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
dstrootcax3	2018 年 4 月 21 日	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13

名前	日付	SHA1 フィンガープリント
certigna	2018 年 4 月 21 日	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
digicerthighassurancceevrootca	2018 年 4 月 21 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
soneraclass2rootca	2018 年 4 月 21 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
trustcorrootcertca2	2018 年 4 月 21 日	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
usertrustrsacertificationauthority	2018 年 4 月 21 日	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
trustcorrootcertca1	2018 年 4 月 21 日	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
geotrustuniversalca	2018 年 4 月 21 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
certsignrootca	2018 年 4 月 21 日	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
amazonrootca4	2018 年 4 月 21 日	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
amazonrootca3	2018 年 4 月 21 日	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
amazonrootca2	2018 年 4 月 21 日	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
verisignuniversalrootcertificationauthority	2018 年 4 月 21 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54

名前	日付	SHA1 フィンガープリント
amazonrootca1	2018 年 4 月 21 日	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
networksolutionscertificate authority	2018 年 4 月 21 日	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
thawteprimaryrootcag3	2018 年 4 月 21 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
affirmtrustnetworking	2018 年 4 月 21 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
thawteprimaryrootcag2	2018 年 4 月 21 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
trustcoreca1	2018 年 4 月 21 日	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
deutschetelekomrootca2	2018 年 4 月 21 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
godaddyrootcertificateauthorityg2	2018 年 4 月 21 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
entrustrootcertificationauthorityec1	2018 年 4 月 21 日	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
szafirrootca2	2018 年 4 月 21 日	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE
tubitakkamusmsslkoksertifikasisurum1	2018 年 4 月 21 日	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA

名前	日付	SHA1 フィンガープリント
buypassclass3rootca	2018 年 4 月 21 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
comodorsacertificationauthority	2018 年 4 月 21 日	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
netlockaranyclassgoldfotanusitvany	2018 年 4 月 21 日	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
securitycommunicationrootca2	2018 年 4 月 21 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
dtrustrootclass3ca2ev2009	2018 年 4 月 21 日	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
starfieldclass2ca	2018 年 4 月 21 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
pscprocert	2018 年 4 月 21 日	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
actalisauthenticationrootca	2018 年 4 月 21 日	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
staatdernederlandenrootcag3	2018 年 4 月 21 日	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
cfcaevroot	2018 年 4 月 21 日	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
digicerttrustedrootg4	2018 年 4 月 21 日	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
staatdernederlandenrootcag2	2018 年 4 月 21 日	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
securitycommunicationevrootca1	2018 年 4 月 21 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D

名前	日付	SHA1 フィンガープリント
globalsignrootcar3	2018 年 4 月 21 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
globalsignrootcar2	2018 年 4 月 21 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
certumtrustednetworkca2	2018 年 4 月 21 日	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
acraizfnmtrcm	2018 年 4 月 21 日	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
hellenicacademicandresearchinstitutionseccrootca2015	2018 年 4 月 21 日	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
certplusrootcag2	2018 年 4 月 21 日	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:CC:1A
twcarootcertificationauthority	2018 年 4 月 21 日	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
twcaglobalrootca	2018 年 4 月 21 日	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
certplusrootcag1	2018 年 4 月 21 日	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:F7:66
geotrustuniversalca2	2018 年 4 月 21 日	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
baltimorecybertrustroot	2018 年 4 月 21 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
buypassclass2rootca	2018 年 4 月 21 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99

名前	日付	SHA1 フィンガープリント
certumtrustednetworkca	2018 年 4 月 21 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
digicertassuredidrootg3	2018 年 4 月 21 日	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
digicertassuredidrootg2	2018 年 4 月 21 日	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F
isrgrootx1	2018 年 4 月 21 日	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
entrustnetpremium2048secureserverca	2018 年 4 月 21 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
certplusclass2primaryca	2018 年 4 月 21 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
digicertglobalrootca	2018 年 4 月 21 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
entrustrootcertificationauthorityg2	2018 年 4 月 21 日	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
starfieldservicesrootcertificateauthorityg2	2018 年 4 月 21 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
thawteprimaryrootca	2018 年 4 月 21 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
atostrustedroot2011	2018 年 4 月 21 日	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
geotrustglobalca	2018 年 4 月 21 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12

名前	日付	SHA1 フィンガープリント
luxtrustglobalroot2	2018 年 4 月 21 日	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5F:3F
etugracertificationauthority	2018 年 4 月 21 日	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
visaecommerceroot	2018 年 4 月 21 日	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
quovadisrootca	2018 年 4 月 21 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
identrustcommercialrootca1	2018 年 4 月 21 日	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
staatdernederlandenevrootca	2018 年 4 月 21 日	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
ttelesecglobalrootclass3	2018 年 4 月 21 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
ttelesecglobalrootclass2	2018 年 4 月 21 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
comodocertificationauthority	2018 年 4 月 21 日	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
securitycommunicationrootca	2018 年 4 月 21 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
quovadisrootca3g3	2018 年 4 月 21 日	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D
xrampglobalcaroot	2018 年 4 月 21 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
securesignrootca11	2018 年 4 月 21 日	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3

名前	日付	SHA1 フィンガープリント
affirmtrustpremium	2018 年 4 月 21 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
globalsignrootca	2018 年 4 月 21 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
swisssigngoldcag2	2018 年 4 月 21 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
quovadisrootca2g3	2018 年 4 月 21 日	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
affirmtrustpremiumecc	2018 年 4 月 21 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
geotrustprimarycertificationauthority	2018 年 4 月 21 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
quovadisrootca1g3	2018 年 4 月 21 日	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67
hongkongpostrootca1	2018 年 4 月 21 日	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
usertrustecccertificationauthority	2018 年 4 月 21 日	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0
cybertrustglobalroot	2018 年 4 月 21 日	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
godaddyclass2ca	2018 年 4 月 21 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4

名前	日付	SHA1 フィンガープリント
hellenicacademicandresearchinstitutionsrootca2015	2018 年 4 月 21 日	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
ecacc	2018 年 4 月 21 日	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:F2:E8
hellenicacademicandresearchinstitutionsrootca2011	2018 年 4 月 21 日	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
verisignclass3publicprimarycertificationauthorityg5	2018 年 4 月 21 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
verisignclass3publicprimarycertificationauthorityg4	2018 年 4 月 21 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3publicprimarycertificationauthorityg3	2018 年 4 月 21 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
trustisfpsrootca	2018 年 4 月 21 日	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
epkirootcertificationauthority	2018 年 4 月 21 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
globalchambersignroot2008	2018 年 4 月 21 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C

名前	日付	SHA1 フィンガープリント
camerfirmachambersofcommerceroot	2018 年 4 月 21 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert81.pem	2014 年 3 月 13 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
mozillacert99.pem	2014 年 3 月 13 日	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
mozillacert145.pem	2014 年 3 月 13 日	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:3A:04
mozillacert37.pem	2014 年 3 月 13 日	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert4.pem	2014 年 3 月 13 日	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B
mozillacert70.pem	2014 年 3 月 13 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
mozillacert88.pem	2014 年 3 月 13 日	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
mozillacert134.pem	2014 年 3 月 13 日	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
mozillacert26.pem	2014 年 3 月 13 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
mozillacert77.pem	2014 年 3 月 13 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
mozillacert123.pem	2014 年 3 月 13 日	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10:DD:6B:DF:99:72:2C:96:E5
mozillacert15.pem	2014 年 3 月 13 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB

名前	日付	SHA1 フィンガープリント
mozillacert66.pem	2014 年 3 月 13 日	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34
mozillacert112.pem	2014 年 3 月 13 日	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
mozillacert55.pem	2014 年 3 月 13 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
mozillacert101.pem	2014 年 3 月 13 日	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
mozillacert119.pem	2014 年 3 月 13 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
mozillacert44.pem	2014 年 3 月 13 日	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
mozillacert108.pem	2014 年 3 月 13 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
mozillacert95.pem	2014 年 3 月 13 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert141.pem	2014 年 3 月 13 日	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
mozillacert33.pem	2014 年 3 月 13 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
mozillacert0.pem	2014 年 3 月 13 日	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
mozillacert84.pem	2014 年 3 月 13 日	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2
mozillacert130.pem	2014 年 3 月 13 日	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E

名前	日付	SHA1 フィンガープリント
mozillacert148.pem	2014 年 3 月 13 日	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
mozillacert22.pem	2014 年 3 月 13 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
mozillacert7.pem	2014 年 3 月 13 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
mozillacert73.pem	2014 年 3 月 13 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
mozillacert137.pem	2014 年 3 月 13 日	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A1:D1
mozillacert11.pem	2014 年 3 月 13 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
mozillacert29.pem	2014 年 3 月 13 日	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
mozillacert62.pem	2014 年 3 月 13 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert126.pem	2014 年 3 月 13 日	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
mozillacert18.pem	2014 年 3 月 13 日	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
mozillacert51.pem	2014 年 3 月 13 日	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
mozillacert69.pem	2014 年 3 月 13 日	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
mozillacert115.pem	2014 年 3 月 13 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9

名前	日付	SHA1 フィンガープリント
mozillacert40.pem	2014 年 3 月 13 日	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
mozillacert58.pem	2014 年 3 月 13 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
mozillacert104.pem	2014 年 3 月 13 日	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E
mozillacert91.pem	2014 年 3 月 13 日	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
mozillacert47.pem	2014 年 3 月 13 日	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
mozillacert80.pem	2014 年 3 月 13 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert98.pem	2014 年 3 月 13 日	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
mozillacert144.pem	2014 年 3 月 13 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
mozillacert36.pem	2014 年 3 月 13 日	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D
mozillacert3.pem	2014 年 3 月 13 日	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
mozillacert87.pem	2014 年 3 月 13 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert133.pem	2014 年 3 月 13 日	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
mozillacert25.pem	2014 年 3 月 13 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5

名前	日付	SHA1 フィンガープリント
mozillacert76.pem	2014 年 3 月 13 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert122.pem	2014 年 3 月 13 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
mozillacert14.pem	2014 年 3 月 13 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
mozillacert65.pem	2014 年 3 月 13 日	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:35:FB
mozillacert111.pem	2014 年 3 月 13 日	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
mozillacert129.pem	2014 年 3 月 13 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
mozillacert54.pem	2014 年 3 月 13 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
mozillacert100.pem	2014 年 3 月 13 日	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
mozillacert118.pem	2014 年 3 月 13 日	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45
mozillacert151.pem	2014 年 3 月 13 日	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
mozillacert43.pem	2014 年 3 月 13 日	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
mozillacert107.pem	2014 年 3 月 13 日	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
mozillacert94.pem	2014 年 3 月 13 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99

名前	日付	SHA1 フィンガープリント
mozillacert140.pem	2014 年 3 月 13 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert32.pem	2014 年 3 月 13 日	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C
mozillacert83.pem	2014 年 3 月 13 日	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:EA:46
mozillacert147.pem	2014 年 3 月 13 日	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
mozillacert21.pem	2014 年 3 月 13 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
mozillacert39.pem	2014 年 3 月 13 日	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
mozillacert6.pem	2014 年 3 月 13 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
mozillacert72.pem	2014 年 3 月 13 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
mozillacert136.pem	2014 年 3 月 13 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert10.pem	2014 年 3 月 13 日	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert28.pem	2014 年 3 月 13 日	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
mozillacert61.pem	2014 年 3 月 13 日	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
mozillacert79.pem	2014 年 3 月 13 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27

名前	日付	SHA1 フィンガープリント
mozillacert125.pem	2014 年 3 月 13 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert17.pem	2014 年 3 月 13 日	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert50.pem	2014 年 3 月 13 日	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
mozillacert68.pem	2014 年 3 月 13 日	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
mozillacert114.pem	2014 年 3 月 13 日	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert57.pem	2014 年 3 月 13 日	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
mozillacert103.pem	2014 年 3 月 13 日	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert90.pem	2014 年 3 月 13 日	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
mozillacert46.pem	2014 年 3 月 13 日	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
mozillacert97.pem	2014 年 3 月 13 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
mozillacert143.pem	2014 年 3 月 13 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
mozillacert35.pem	2014 年 3 月 13 日	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
mozillacert2.pem	2014 年 3 月 13 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A

名前	日付	SHA1 フィンガープリント
mozillacert86.pem	2014 年 3 月 13 日	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
mozillacert132.pem	2014 年 3 月 13 日	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert24.pem	2014 年 3 月 13 日	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
mozillacert9.pem	2014 年 3 月 13 日	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
mozillacert75.pem	2014 年 3 月 13 日	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
mozillacert121.pem	2014 年 3 月 13 日	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
mozillacert139.pem	2014 年 3 月 13 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert13.pem	2014 年 3 月 13 日	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
mozillacert64.pem	2014 年 3 月 13 日	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
mozillacert110.pem	2014 年 3 月 13 日	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert128.pem	2014 年 3 月 13 日	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D
mozillacert53.pem	2014 年 3 月 13 日	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
mozillacert117.pem	2014 年 3 月 13 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74

名前	日付	SHA1 フィンガープリント
mozillacert150.pem	2014 年 3 月 13 日	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
mozillacert42.pem	2014 年 3 月 13 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
mozillacert106.pem	2014 年 3 月 13 日	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
mozillacert93.pem	2014 年 3 月 13 日	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17
mozillacert31.pem	2014 年 3 月 13 日	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
mozillacert49.pem	2014 年 3 月 13 日	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71
mozillacert82.pem	2014 年 3 月 13 日	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
mozillacert146.pem	2014 年 3 月 13 日	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A
mozillacert20.pem	2014 年 3 月 13 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert38.pem	2014 年 3 月 13 日	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36
mozillacert5.pem	2014 年 3 月 13 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
mozillacert71.pem	2014 年 3 月 13 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert89.pem	2014 年 3 月 13 日	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D

名前	日付	SHA1 フィンガープリント
mozillacert135.pem	2014 年 3 月 13 日	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert27.pem	2014 年 3 月 13 日	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
mozillacert60.pem	2014 年 3 月 13 日	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert78.pem	2014 年 3 月 13 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert124.pem	2014 年 3 月 13 日	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert16.pem	2014 年 3 月 13 日	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
mozillacert67.pem	2014 年 3 月 13 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
mozillacert113.pem	2014 年 3 月 13 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
mozillacert56.pem	2014 年 3 月 13 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
mozillacert102.pem	2014 年 3 月 13 日	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
mozillacert45.pem	2014 年 3 月 13 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert109.pem	2014 年 3 月 13 日	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
mozillacert96.pem	2014 年 3 月 13 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1

名前	日付	SHA1 フィンガープリント
mozillacert142.pem	2014 年 3 月 13 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
mozillacert34.pem	2014 年 3 月 13 日	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
mozillacert1.pem	2014 年 3 月 13 日	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
mozillacert85.pem	2014 年 3 月 13 日	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
mozillacert131.pem	2014 年 3 月 13 日	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
mozillacert149.pem	2014 年 3 月 13 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert23.pem	2014 年 3 月 13 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
mozillacert8.pem	2014 年 3 月 13 日	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:6A:0F
mozillacert74.pem	2014 年 3 月 13 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert120.pem	2014 年 3 月 13 日	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
mozillacert138.pem	2014 年 3 月 13 日	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
mozillacert12.pem	2014 年 3 月 13 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
mozillacert63.pem	2014 年 3 月 13 日	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E

名前	日付	SHA1 フィンガープリント
mozillacert127.pem	2014 年 3 月 13 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert19.pem	2014 年 3 月 13 日	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
mozillacert52.pem	2014 年 3 月 13 日	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F
mozillacert116.pem	2014 年 3 月 13 日	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert41.pem	2014 年 3 月 13 日	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
mozillacert59.pem	2014 年 3 月 13 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
mozillacert105.pem	2014 年 3 月 13 日	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC
mozillacert92.pem	2014 年 3 月 13 日	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0
mozillacert30.pem	2014 年 3 月 13 日	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
mozillacert48.pem	2014 年 3 月 13 日	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC
verisignc4g2.pem	2014 年 3 月 20 日	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52
verisignc2g3.pem	2014 年 3 月 20 日	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
verisignc1g6.pem	2014 年 12 月 31 日	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64

名前	日付	SHA1 フィンガープリント
verisignc2g2.pem	2014 年 3 月 20 日	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
verisignroot.pem	2014 年 3 月 20 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
verisignc2g1.pem	2014 年 3 月 20 日	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:14:2A
verisignc3g5.pem	2014 年 3 月 20 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
verisignc1g3.pem	2014 年 3 月 20 日	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
verisignc3g4.pem	2014 年 3 月 20 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignc1g2.pem	2014 年 3 月 20 日	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
verisignc3g3.pem	2014 年 3 月 20 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
verisignc1g1.pem	2014 年 3 月 20 日	90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:55:6F
verisignc3g2.pem	2014 年 3 月 20 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
verisignc3g1.pem	2014 年 3 月 20 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
verisignc2g6.pem	2014 年 12 月 31 日	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F
verisignc4g3.pem	2014 年 3 月 20 日	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D

名前	日付	SHA1 フィンガープリント
gdroot-g2.pem	2014 年 12 月 31 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
gd-class2-root.pem	2014 年 12 月 31 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
gd_bundle-g2.pem	2014 年 12 月 31 日	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8
dstacescax6	2018 年 6 月 18 日	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
gd_bundle-g2.pem	2018 年 6 月 18 日	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8
verisignc4g3.pem	2018 年 6 月 18 日	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
swisssignplatinumg2ca	2018 年 4 月 21 日	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66
geotrustprimarycertificationauthorityg3	2018 年 6 月 18 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycertificationauthorityg2	2018 年 6 月 18 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
buypassclass2rootca	2018 年 6 月 18 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
camerfirmachambersofcommerceroot	2018 年 6 月 18 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert20.pem	2018 年 6 月 18 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61

名前	日付	SHA1 フィンガープリント
mozillacert12.pem	2018 年 6 月 18 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
mozillacert90.pem	2018 年 6 月 18 日	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
mozillacert82.pem	2018 年 6 月 18 日	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
mozillacert140.pem	2018 年 6 月 18 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert74.pem	2018 年 6 月 18 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert132.pem	2018 年 6 月 18 日	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert66.pem	2018 年 6 月 18 日	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34
mozillacert124.pem	2018 年 6 月 18 日	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert58.pem	2018 年 6 月 18 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
securitycommunicationrootca2	2018 年 6 月 18 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert116.pem	2018 年 6 月 18 日	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert108.pem	2018 年 6 月 18 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
certigna	2018 年 6 月 18 日	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97

名前	日付	SHA1 フィンガープリント
mozillacert3.pem	2018 年 6 月 18 日	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
verisignc1g1.pem	2018 年 6 月 18 日	90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:55:6F
verisignc4g2.pem	2018 年 6 月 18 日	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52
deutschetelekomrootca2	2018 年 6 月 18 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
starfieldrootg2ca	2018 年 4 月 21 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
comodoecccertificationauthority	2018 年 6 月 18 日	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
digicertglobalrootg3	2018 年 6 月 18 日	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E
digicertglobalrootg2	2018 年 6 月 18 日	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
mozillacert11.pem	2018 年 6 月 18 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
mozillacert81.pem	2018 年 6 月 18 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
mozillacert73.pem	2018 年 6 月 18 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
szafirrootca2	2018 年 6 月 18 日	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE
mozillacert131.pem	2018 年 6 月 18 日	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79

名前	日付	SHA1 フィンガープリント
ecacc	2018 年 6 月 18 日	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:F2:E8
mozillacert65.pem	2018 年 6 月 18 日	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:35:FB
turktrustelektroniksertifikahizmetsaglayicisih5	2018 年 6 月 18 日	C4:18:F6:4D:46:D1:DF:00:3D:27:30:13:72:43:A9:12:11:C6:75:FB
mozillacert123.pem	2018 年 6 月 18 日	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10:DD:6B:DF:99:72:2C:96:E5
mozillacert57.pem	2018 年 6 月 18 日	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
mozillacert115.pem	2018 年 6 月 18 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert49.pem	2018 年 6 月 18 日	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71
mozillacert107.pem	2018 年 6 月 18 日	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
verisignclass3g4ca	2018 年 4 月 21 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
securetrustca	2018 年 6 月 18 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
mozillacert2.pem	2018 年 6 月 18 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
buypassclass2ca	2018 年 4 月 21 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99

名前	日付	SHA1 フィンガープリント
secomscrootca2	2018 年 4 月 21 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
secomscrootca1	2018 年 4 月 21 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
trustisfpsrootca	2018 年 6 月 18 日	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
hongkongpostrootca1	2018 年 6 月 18 日	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
certsignrootca	2018 年 6 月 18 日	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
geotrustprimaryca	2018 年 4 月 21 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
twcaglobalrootca	2018 年 6 月 18 日	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
camerfirmachambersca	2018 年 4 月 21 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
mozillacert10.pem	2018 年 6 月 18 日	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert80.pem	2018 年 6 月 18 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert72.pem	2018 年 6 月 18 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
comodoaaaca	2018 年 4 月 21 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert130.pem	2018 年 6 月 18 日	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E

名前	日付	SHA1 フィンガープリント
mozillacert64.pem	2018 年 6 月 18 日	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
mozillacert122.pem	2018 年 6 月 18 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
mozillacert56.pem	2018 年 6 月 18 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
equifaxsecureebusinessca1	2018 年 4 月 21 日	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
camerfirmachambersignca	2018 年 4 月 21 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert114.pem	2018 年 6 月 18 日	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert48.pem	2018 年 6 月 18 日	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC
pscprocert	2018 年 6 月 18 日	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert106.pem	2018 年 6 月 18 日	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
mozillacert1.pem	2018 年 6 月 18 日	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
eecertificationcentrerootca	2018 年 6 月 18 日	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
digicertglobalrootca	2018 年 6 月 18 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
thawteprimaryrootcag3	2018 年 6 月 18 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2

名前	日付	SHA1 フィンガープリント
thawteprimaryrootcag2	2018 年 6 月 18 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
entrustrootcertificationauthorityec1	2018 年 6 月 18 日	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
valicertclass2ca	2018 年 4 月 21 日	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
globalchambersignroot2008	2018 年 6 月 18 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
amazonrootca4	2018 年 6 月 18 日	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
gd-class2-root.pem	2018 年 6 月 18 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
amazonrootca3	2018 年 6 月 18 日	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
amazonrootca2	2018 年 6 月 18 日	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
securitycommunicationrootca	2018 年 6 月 18 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
amazonrootca1	2018 年 6 月 18 日	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
acraizfnmtrcm	2018 年 6 月 18 日	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
quovadisrootca3g3	2018 年 6 月 18 日	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D

名前	日付	SHA1 フィンガープリント
certplusrootcag2	2018 年 6 月 18 日	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:CC:1A
certplusrootcag1	2018 年 6 月 18 日	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:F7:66
mozillacert71.pem	2018 年 6 月 18 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert63.pem	2018 年 6 月 18 日	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
mozillacert121.pem	2018 年 6 月 18 日	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
ttelesecglobalrootclass3ca	2018 年 4 月 21 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
mozillacert55.pem	2018 年 6 月 18 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
mozillacert113.pem	2018 年 6 月 18 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
baltimorecybertrustca	2018 年 4 月 21 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
mozillacert47.pem	2018 年 6 月 18 日	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
mozillacert105.pem	2018 年 6 月 18 日	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC
mozillacert39.pem	2018 年 6 月 18 日	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
usertrustecccertificationauthority	2018 年 6 月 18 日	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0

名前	日付	SHA1 フィンガープリント
mozillacert0.pem	2018 年 6 月 18 日	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
securitycommunicationevrootca1	2018 年 6 月 18 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
verisignc3g5.pem	2018 年 6 月 18 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
globalsignr3ca	2018 年 4 月 21 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
trustcoreca1	2018 年 6 月 18 日	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
equifaxsecureglobalebusinessca1	2018 年 4 月 21 日	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
geotrustuniversalca	2018 年 6 月 18 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
affirmtrustpremiumca	2018 年 4 月 21 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
staatdernederlandenrootcag3	2018 年 6 月 18 日	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
staatdernederlandenrootcag2	2018 年 6 月 18 日	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
mozillacert70.pem	2018 年 6 月 18 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
secomevrootca1	2018 年 4 月 21 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
geotrustglobalca	2018 年 6 月 18 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12

名前	日付	SHA1 フィンガープリント
mozillacert62.pem	2018 年 6 月 18 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert120.pem	2018 年 6 月 18 日	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
mozillacert54.pem	2018 年 6 月 18 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
mozillacert112.pem	2018 年 6 月 18 日	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
mozillacert46.pem	2018 年 6 月 18 日	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
swisssigngoldcag2	2018 年 6 月 18 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert104.pem	2018 年 6 月 18 日	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E
mozillacert38.pem	2018 年 6 月 18 日	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36
certplusclass3pprimaryca	2018 年 4 月 21 日	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
entrustrootcertificationauthorityg2	2018 年 6 月 18 日	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
godaddyrootg2ca	2018 年 4 月 21 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
cfcaevroot	2018 年 6 月 18 日	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
verisignc3g4.pem	2018 年 6 月 18 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A

名前	日付	SHA1 フィンガープリント
geotrustuniversalca2	2018 年 6 月 18 日	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
starfieldservicesrootg2ca	2018 年 4 月 21 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
digicerthighassuranceevrootca	2018 年 6 月 18 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
entrustnetpremium2048secureserverca	2018 年 6 月 18 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
camerfirmaglobalchambersignroot	2018 年 6 月 18 日	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
verisignclass3g3ca	2018 年 4 月 21 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
godaddyclass2ca	2018 年 6 月 18 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
mozillacert61.pem	2018 年 6 月 18 日	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
mozillacert53.pem	2018 年 6 月 18 日	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
atostrustedroot2011	2018 年 6 月 18 日	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert111.pem	2018 年 6 月 18 日	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
staatdernederlandenevrootca	2018 年 6 月 18 日	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
mozillacert45.pem	2018 年 6 月 18 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0

名前	日付	SHA1 フィンガープリント
mozillacert103.pem	2018 年 6 月 18 日	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert37.pem	2018 年 6 月 18 日	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert29.pem	2018 年 6 月 18 日	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
izenpecom	2018 年 6 月 18 日	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
comodorsacertificationauthority	2018 年 6 月 18 日	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
mozillacert99.pem	2018 年 6 月 18 日	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
mozillacert149.pem	2018 年 6 月 18 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
utnuserfirstobjectca	2018 年 4 月 21 日	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46
verisignc3g3.pem	2018 年 6 月 18 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
dstrootcax3	2018 年 6 月 18 日	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
addtrustexternalroot	2018 年 6 月 18 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
certumtrustednetworkca	2018 年 6 月 18 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
affirmtrustpremiumecc	2018 年 6 月 18 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB

名前	日付	SHA1 フィンガープリント
starfieldclass2ca	2018 年 6 月 18 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
actalisauthenticationrootca	2018 年 6 月 18 日	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
verisignclass2g3ca	2018 年 4 月 21 日	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
isrgrootx1	2018 年 6 月 18 日	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
godaddyrootcertificateauthorityg2	2018 年 6 月 18 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
mozillacert60.pem	2018 年 6 月 18 日	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
chunghwaepkirootca	2018 年 4 月 21 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert52.pem	2018 年 6 月 18 日	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F
microseceszignorootca2009	2018 年 6 月 18 日	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
securesignrootca11	2018 年 6 月 18 日	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert110.pem	2018 年 6 月 18 日	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert44.pem	2018 年 6 月 18 日	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
mozillacert102.pem	2018 年 6 月 18 日	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83

名前	日付	SHA1 フィンガープリント
mozillacert36.pem	2018 年 6 月 18 日	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D
mozillacert28.pem	2018 年 6 月 18 日	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
baltimorecybertrustroot	2018 年 6 月 18 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
amzninternalrootca	2008 年 12 月 12 日	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06
mozillacert98.pem	2018 年 6 月 18 日	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
mozillacert148.pem	2018 年 6 月 18 日	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
verisignc3g2.pem	2018 年 6 月 18 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
quovadisrootca2g3	2018 年 6 月 18 日	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
geotrustprimarycertificatio nauthority	2018 年 6 月 18 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
opentrustrootcag3	2018 年 6 月 18 日	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8A:A6
opentrustrootcag2	2018 年 6 月 18 日	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	2018 年 6 月 18 日	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:D5:7E

名前	日付	SHA1 フィンガープリント
verisignclass3ca	2018 年 4 月 21 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
globalsignca	2018 年 4 月 21 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
ttelesecglobalrootclass2ca	2018 年 4 月 21 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
verisignclass1g3ca	2018 年 4 月 21 日	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
verisignuniversalrootca	2018 年 4 月 21 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
soneraclass2ca	2018 年 4 月 21 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
starfieldservicesrootcertificateauthorityg2	2018 年 6 月 18 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert51.pem	2018 年 6 月 18 日	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
mozillacert43.pem	2018 年 6 月 18 日	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
mozillacert101.pem	2018 年 6 月 18 日	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
mozillacert35.pem	2018 年 6 月 18 日	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
globalsignr2ca	2018 年 4 月 21 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE

名前	日付	SHA1 フィンガープリント
mozillacert27.pem	2018 年 6 月 18 日	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
affirmtrustpremium	2018 年 6 月 18 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert19.pem	2018 年 6 月 18 日	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
mozillacert97.pem	2018 年 6 月 18 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
netlockaranyclassgoldfotanusitvany	2018 年 6 月 18 日	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
mozillacert89.pem	2018 年 6 月 18 日	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
verisignroot.pem	2018 年 6 月 18 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
mozillacert147.pem	2018 年 6 月 18 日	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
aolrootca2	2018 年 4 月 21 日	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
cia-crt-g3-01-ca	2016 年 11 月 23 日	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
aolrootca1	2018 年 4 月 21 日	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignc3g1.pem	2018 年 6 月 18 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert139.pem	2018 年 6 月 18 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9

名前	日付	SHA1 フィンガープリント
soneraclass2rootca	2018 年 6 月 18 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
swisssignsilverg2ca	2018 年 4 月 21 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
thawteprimaryrootca	2018 年 6 月 18 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
gdcatrustauthr5root	2018 年 6 月 18 日	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
trustcenterclass4caii	2018 年 4 月 21 日	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
usertrustrsacertificationauthority	2018 年 6 月 18 日	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
digicertassuredidrootg3	2018 年 6 月 18 日	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
digicertassuredidrootg2	2018 年 6 月 18 日	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F
mozillacert50.pem	2018 年 6 月 18 日	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
mozillacert42.pem	2018 年 6 月 18 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
mozillacert100.pem	2018 年 6 月 18 日	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
mozillacert34.pem	2018 年 6 月 18 日	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
affirmtrustcommercialca	2018 年 4 月 21 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7

名前	日付	SHA1 フィンガープリント
mozillacert26.pem	2018 年 6 月 18 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
globalsigneccrootcar5	2018 年 6 月 18 日	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
globalsigneccrootcar4	2018 年 6 月 18 日	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
buypassclass3rootca	2018 年 6 月 18 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert18.pem	2018 年 6 月 18 日	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
mozillacert96.pem	2018 年 6 月 18 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
verisignc2g6.pem	2018 年 6 月 18 日	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F
secomvalicertclass1ca	2018 年 4 月 21 日	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert88.pem	2018 年 6 月 18 日	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
accvraiz1	2018 年 6 月 18 日	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert146.pem	2018 年 6 月 18 日	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A
mozillacert138.pem	2018 年 6 月 18 日	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
verisignclass3g2ca	2018 年 4 月 21 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F

名前	日付	SHA1 フィンガープリント
dtrustrootclass3ca2ev2009	2018 年 6 月 18 日	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
xrampglobalca	2018 年 4 月 21 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
mozillacert9.pem	2018 年 6 月 18 日	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
verisignuniversalrootcertificationauthority	2018 年 6 月 18 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
tubitakkamusmsslkoksertifikasisurum1	2018 年 6 月 18 日	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
mozillacert41.pem	2018 年 6 月 18 日	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
mozillacert33.pem	2018 年 6 月 18 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
mozillacert25.pem	2018 年 6 月 18 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
mozillacert17.pem	2018 年 6 月 18 日	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert95.pem	2018 年 6 月 18 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
affirmtrustpremiumeccca	2018 年 4 月 21 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert87.pem	2018 年 6 月 18 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74

名前	日付	SHA1 フィンガープリント
mozillacert145.pem	2018 年 6 月 18 日	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:3A:04
mozillacert79.pem	2018 年 6 月 18 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert137.pem	2018 年 6 月 18 日	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A1:D1
digicertassuredidrootca	2018 年 6 月 18 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
addtrustqualifiedca	2018 年 4 月 21 日	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert129.pem	2018 年 6 月 18 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
verisignclass2g2ca	2018 年 4 月 21 日	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
baltimorecodesigningca	2018 年 4 月 21 日	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D
luxtrustglobalroot2	2018 年 6 月 18 日	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5F:3F
visaecommerceroot	2018 年 6 月 18 日	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
oistewisekeyglobalrootgbca	2018 年 6 月 18 日	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
mozillacert8.pem	2018 年 6 月 18 日	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:6A:0F
comodocertificationauthority	2018 年 6 月 18 日	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B

名前	日付	SHA1 フィンガープリント
cia-crt-g3-02-ca	2016 年 11 月 23 日	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
verisignc1g6.pem	2018 年 6 月 18 日	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
trustcenterclass2caii	2018 年 4 月 21 日	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
quovadisrootca1g3	2018 年 6 月 18 日	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67
mozillacert40.pem	2018 年 6 月 18 日	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
cadisigrootr2	2018 年 6 月 18 日	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
cadisigrootr1	2018 年 6 月 18 日	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
mozillacert32.pem	2018 年 6 月 18 日	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C
utndatacorpsgcca	2018 年 4 月 21 日	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
mozillacert24.pem	2018 年 6 月 18 日	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
addtrustclass1ca	2018 年 4 月 21 日	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
mozillacert16.pem	2018 年 6 月 18 日	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
affirmtrustnetworkingca	2018 年 4 月 21 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F

名前	日付	SHA1 フィンガープリント
mozillacert94.pem	2018 年 6 月 18 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
mozillacert86.pem	2018 年 6 月 18 日	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
mozillacert144.pem	2018 年 6 月 18 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
mozillacert78.pem	2018 年 6 月 18 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert136.pem	2018 年 6 月 18 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert128.pem	2018 年 6 月 18 日	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D
verisignclass1g2ca	2018 年 4 月 21 日	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
hellenicacademicandresearchinstitutionsrootca2015	2018 年 6 月 18 日	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
soneraclass1ca	2018 年 4 月 21 日	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
hellenicacademicandresearchinstitutionsrootca2011	2018 年 6 月 18 日	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
certumtrustednetworkca2	2018 年 6 月 18 日	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92

名前	日付	SHA1 フィンガープリント
equifaxsecureca	2018 年 4 月 21 日	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
thawteserverca	2018 年 4 月 21 日	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
mozillacert7.pem	2018 年 6 月 18 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
affirmtrustnetworking	2018 年 6 月 18 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
deprecateditsecca	2012 年 1 月 27 日	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
globalsignrootcar3	2018 年 6 月 18 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
globalsignrootcar2	2018 年 6 月 18 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
quovadisrootca	2018 年 6 月 18 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert31.pem	2018 年 6 月 18 日	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
entrustrootcertificationauthority	2018 年 6 月 18 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert23.pem	2018 年 6 月 18 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
mozillacert15.pem	2018 年 6 月 18 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
verisignc2g3.pem	2018 年 6 月 18 日	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11

名前	日付	SHA1 フィンガープリント
mozillacert93.pem	2018 年 6 月 18 日	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17
mozillacert151.pem	2018 年 6 月 18 日	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
mozillacert85.pem	2018 年 6 月 18 日	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
certplusclass2primaryca	2018 年 6 月 18 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
mozillacert143.pem	2018 年 6 月 18 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
mozillacert77.pem	2018 年 6 月 18 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
mozillacert135.pem	2018 年 6 月 18 日	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert69.pem	2018 年 6 月 18 日	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
mozillacert127.pem	2018 年 6 月 18 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert119.pem	2018 年 6 月 18 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
geotrustprimarycag3	2018 年 4 月 21 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
identrustpublicsectorrootca1	2018 年 6 月 18 日	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
geotrustprimarycag2	2018 年 4 月 21 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0

名前	日付	SHA1 フィンガープリント
trustcorrootcertca2	2018 年 6 月 18 日	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
mozillacert6.pem	2018 年 6 月 18 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
trustcorrootcertca1	2018 年 6 月 18 日	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
networksolutionscertificate authority	2018 年 6 月 18 日	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
twcarootcertificationauthority	2018 年 6 月 18 日	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
addtrustexternalca	2018 年 4 月 21 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
verisignclass3g5ca	2018 年 4 月 21 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
autoridaddecertificacionfirmaprofesionalcifa62634068	2018 年 6 月 18 日	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
hellenicacademicandresearchinstitutionseccrootca2015	2018 年 6 月 18 日	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
verisigntsaca	2018 年 4 月 21 日	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
utnuserfirsthardwareca	2018 年 4 月 21 日	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7

名前	日付	SHA1 フィンガープリント
identrustcommercialrootca1	2018 年 6 月 18 日	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
dtrustrootclass3ca22009	2018 年 6 月 18 日	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
epkirootcertificationauthority	2018 年 6 月 18 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert30.pem	2018 年 6 月 18 日	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
teliasonerarootcav1	2018 年 6 月 18 日	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
buypassclass3ca	2018 年 4 月 21 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert22.pem	2018 年 6 月 18 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
mozillacert14.pem	2018 年 6 月 18 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
verisignc2g2.pem	2018 年 6 月 18 日	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
certumca	2018 年 4 月 21 日	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert92.pem	2018 年 6 月 18 日	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0
mozillacert150.pem	2018 年 6 月 18 日	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
mozillacert84.pem	2018 年 6 月 18 日	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2

名前	日付	SHA1 フィンガープリント
ttelesecglobalroot class3	2018 年 6 月 18 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70: 19:9D:2A:BE:11:E3:81:D1
globalsignrootca	2018 年 6 月 18 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81: F2:15:01:52:A4:1D:82:9C
ttelesecglobalroot class2	2018 年 6 月 18 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62: 32:17:65:CF:17:D8:94:E9
mozillacert142.pem	2018 年 6 月 18 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0: BE:FD:3A:2D:82:75:51:85
mozillacert76.pem	2018 年 6 月 18 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80: DC:E9:6E:2C:C7:B2:78:B7
mozillacert134.pem	2018 年 6 月 18 日	70:17:9B:86:8C:00:A4:FA:60:91:52:22: 3F:9F:3E:32:BD:E0:05:62
mozillacert68.pem	2018 年 6 月 18 日	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07: 5A:9A:E8:00:B7:F7:B6:FA
etugracertificatio nauthority	2018 年 6 月 18 日	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0: 0D:6D:A3:62:8F:C3:52:39
mozillacert126.pem	2018 年 6 月 18 日	25:01:90:19:CF:FB:D9:99:1C:B7:68:25: 74:8D:94:5F:30:93:95:42
keynectisrootca	2018 年 4 月 21 日	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D: BA:EA:E4:A2:D2:D5:CC:97
mozillacert118.pem	2018 年 6 月 18 日	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D: 47:B4:40:CA:D9:0A:19:45
quovadisrootca3	2018 年 6 月 18 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0: BE:FD:3A:2D:82:75:51:85
quovadisrootca2	2018 年 6 月 18 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20: 88:80:48:39:19:93:7C:F7

名前	日付	SHA1 フィンガープリント
mozillacert5.pem	2018 年 6 月 18 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
verisignc1g3.pem	2018 年 6 月 18 日	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
cybertrustglobalroot	2018 年 6 月 18 日	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
amzninternalinfosecag3	2015 年 2 月 27 日	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
starfieldrootcertificateauthorityg2	2018 年 6 月 18 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
entrust2048ca	2018 年 4 月 21 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
swisssignsilvercag2	2018 年 6 月 18 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
affirmtrustcommercial	2018 年 6 月 18 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
certinomisrootca	2018 年 6 月 18 日	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:88:A8
xrampglobalcaroot	2018 年 6 月 18 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
secureglobalca	2018 年 6 月 18 日	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
swisssigngoldg2ca	2018 年 4 月 21 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert21.pem	2018 年 6 月 18 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB

名前	日付	SHA1 フィンガープリント
mozillacert13.pem	2018 年 6 月 18 日	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
verisignc2g1.pem	2018 年 6 月 18 日	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:14:2A
mozillacert91.pem	2018 年 6 月 18 日	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
oistewisekeyglobalrootgaca	2018 年 6 月 18 日	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
mozillacert83.pem	2018 年 6 月 18 日	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:EA:46
entrustevca	2018 年 4 月 21 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert141.pem	2018 年 6 月 18 日	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
mozillacert75.pem	2018 年 6 月 18 日	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
mozillacert133.pem	2018 年 6 月 18 日	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
mozillacert67.pem	2018 年 6 月 18 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
mozillacert125.pem	2018 年 6 月 18 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert59.pem	2018 年 6 月 18 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
thawtepremiumserverca	2018 年 4 月 21 日	E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:66:66

名前	日付	SHA1 フィンガープリント
mozillacert117.pem	2018 年 6 月 18 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
utnuserfirstclientauthemailca	2018 年 4 月 21 日	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
entrustrootcag2	2018 年 4 月 21 日	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
mozillacert109.pem	2018 年 6 月 18 日	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
digicerttrustedrootg4	2018 年 6 月 18 日	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
gdroot-g2.pem	2018 年 6 月 18 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
comodoaaaservicesroot	2018 年 6 月 18 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert4.pem	2018 年 6 月 18 日	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B
verisignclass3publicprimary certificationauthorityg5	2018 年 6 月 18 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
chambersofcommerceroot2008	2018 年 6 月 18 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
verisignclass3publicprimary certificationauthorityg4	2018 年 6 月 18 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A

名前	日付	SHA1 フィンガープリント
verisignclass3publicprimarycertificationauthorityg3	2018 年 6 月 18 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
thawtepersonalfreemailca	2018 年 4 月 21 日	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
verisignc1g2.pem	2018 年 6 月 18 日	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
gtecybertrustglobalca	2018 年 4 月 21 日	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
trustcenteruniversalcai	2018 年 4 月 21 日	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
camerfirmachamberscommerceca	2018 年 4 月 21 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
verisignclass1ca	2018 年 4 月 21 日	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1

AWS AppSync リゾルバーマッピングテンプレートの変更ログ

> **Note**
>
> 現在、主に APPSYNC_JS ランタイムとそのドキュメントをサポートしています。こちらにある APPSYNC_JS ランタイムとそのガイドの使用をご検討ください。

リゾルバーと関数のマッピングテンプレートはバージョニングされています。2018-05-29 などのマッピングテンプレートバージョンは、以下を決定します。

- リクエストテンプレートによって提供されるデータソースリクエスト設定の想定される形式
- リクエストマッピングテンプレートとレスポンスマッピングテンプレートの実行動作

バージョンは YYYY-MM-DD 形式を使用して表され、より後の日付はより新しいバージョンに対応します。このページには、 AWS AppSyncで現在サポートされているマッピングテンプレートのバージョン間の違いが一覧表示されます。

トピック

- バージョンごとのデータソースオペレーションの可用性一覧
- ユニットリゾルバーのマッピングテンプレートのバージョンの変更
- 関数のバージョンの変更
- 2018-05-29
- 2017-02-28

バージョンごとのデータソースオペレーションの可用性一覧

オペレーション/サポートされているバージョン	2017-02-28	2018-05-29
AWS Lambda 呼び出し	あり	あり
AWS Lambda BatchInvoke	あり	あり
none データソース	あり	あり
Amazon OpenSearch GET	あり	あり
Amazon OpenSearch POST	あり	あり
Amazon OpenSearch PUT	あり	あり
Amazon OpenSearch DELE	あり	あり
Amazon OpenSearch GET	あり	あり
DynamoDB GetItem	あり	あり
DynamoDB Scan	あり	あり
DynamoDB Query	あり	あり

オペレーション/サポートされているバージョン	2017-02-28	2018-05-29
DynamoDB DeleteItem	あり	あり
DynamoDB PutItem	あり	あり
DynamoDB BatchGetItem	いいえ	はい
DynamoDB BatchPutItem	いいえ	はい
DynamoDB BatchDeleteItem	いいえ	はい
HTTP	いいえ	はい
Amazon RDS	いいえ	はい

注意: 関数で現在サポートされているのは 2018-05-29 バージョンのみです。

ユニットリゾルバーのマッピングテンプレートのバージョンの変更

ユニットリゾルバーの場合、バージョンはリクエストマッピングテンプレートの本文の一部として指定されています。バージョンを更新するには、`version` フィールドを新しいバージョンに更新するだけです。

たとえば、 AWS Lambda テンプレートのバージョンを更新するには、次のようにします。

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

以下のように version フィールドを `2017-02-28` から `2018-05-29` に更新する必要があります。

```
{
    "version": "2018-05-29", ## Note the version
```

```
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

関数のバージョンの変更

関数の場合、バージョンは function オブジェクトの `functionVersion` フィールドとして指定されています。バージョンを更新するには、`functionVersion` を更新するだけです。注意: 現在、関数では `2018-05-29` のみがサポートされています。

以下に示しているのは、既存の関数バージョンを更新する CLI コマンドの例です。

```
aws appsync update-function \
--api-id REPLACE_WITH_API_ID \
--function-id REPLACE_WITH_FUNCTION_ID \
--data-source-name "PostTable" \
--function-version "2018-05-29" \
--request-mapping-template "{...}" \
--response-mapping-template "\$util.toJson(\$ctx.result)"
```

注意: 関数のリクエストマッピングテンプレートから version フィールドを省略することをお勧めします。それにより、このフィールドが優先されなくなります。関数のリクエストマッピングテンプレート内でバージョンを指定した場合、version の値は `functionVersion` フィールドの値によって書き換えられます。

2018-05-29

動作の変更

- データソースの呼び出しの結果が `null` の場合は、レスポンスマッピングテンプレートが実行されます。
- データソースの呼び出しでエラーが発生した場合、エラーを処理するのはお客様です。レスポンスマッピングテンプレートの評価結果は常に GraphQL レスポンスの `data` ブロック内に挿入されます。

理由

- null の呼び出し結果には意味があり、アプリケーションのユースケースによっては、null の結果を独自の方法で処理することが必要になる場合があります。たとえば、アプリケーションで承認チェックを実行するために、Amazon DynamoDB テーブルにレコードが存在するかどうかを確認するとします。この場合、null の呼び出し結果は、ユーザーが承認されていない可能性があることを意味します。レスポンスマッピングテンプレートを実行することで、承認エラーを発生させることが可能になりました。この動作により、API 設計者はより強力な制御機能を得られます。

以下のレスポンスマッピングテンプレートがあるとします。

```
$util.toJson($ctx.result)
```

以前、2017-02-28 では、$ctx.result から null が返された場合、レスポンスマッピングテンプレートは実行されませんでした。2018-05-29 では、このシナリオを処理できるようになりました。たとえば、以下のように承認エラーを発生させることが可能です。

```
# throw an unauthorized error if the result is null
#if ( $util.isNull($ctx.result) )
    $util.unauthorized()
#end
$util.toJson($ctx.result)
```

注意: データソースから返されるエラーの中には、致命的でないものがあり、予期できないものもあります。そのため、レスポンスマッピングテンプレートには、呼び出しエラーを処理して、無視するか、再発生させるか、別のエラーをスローするかを判断する柔軟性が必要になります。

以下のレスポンスマッピングテンプレートがあるとします。

```
$util.toJson($ctx.result)
```

以前、2017-02-28 では、呼び出しエラーが発生した場合にレスポンスマッピングテンプレートが評価され、結果は GraphQL レスポンスの errors ブロックに自動的に挿入されていました。2018-05-29 では、エラーをどう処理するか、再発生させるか、別のエラーを発生させるか、またはデータを返すときにエラーを付加するかを選択できるようになりました。

呼び出しエラーを再発生させる

以下のレスポンステンプレートでは、データソースから返されたものと同じエラーを発生させます。

```
#if ( $ctx.error )
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

呼び出しエラー ($ctx.error など) の場合、レスポンスは以下のようになります。

```
{
    "data": {
        "getPost": null
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "DynamoDB:ConditionalCheckFailedException",
            "message": "Conditional check failed exception...",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

別のエラーを発生させる

以下のレスポンステンプレートでは、データソースから返されたエラーを処理した後で、カスタムエラーを発生させます。

```
#if ( $ctx.error )
    #if ( $ctx.error.type.equals("ConditionalCheckFailedException") )
        ## we choose here to change the type and message of the error for
 ConditionalCheckFailedExceptions
        $util.error("Error while updating the post, try again. Error:
 $ctx.error.message", "UpdateError")
    #else
        $util.error($ctx.error.message, $ctx.error.type)
    #end
```

```
#end
$util.toJson($ctx.result)
```

呼び出しエラー ($ctx.error など) の場合、レスポンスは以下のようになります。

```
{
    "data": {
        "getPost": null
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "UpdateError",
            "message": "Error while updating the post, try again. Error: Conditional
 check failed exception...",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

データを返すときにエラーを付加する

以下のレスポンステンプレートでは、レスポンス内でデータを返すときに、データソースから返されたものと同じエラーを追加します。これは、部分レスポンスとも呼ばれます。

```
#if ( $ctx.error )
    $util.appendError($ctx.error.message, $ctx.error.type)
    #set($defaultPost = {id: "1", title: 'default post'})
    $util.toJson($defaultPost)
#else
    $util.toJson($ctx.result)
#end
```

呼び出しエラー ($ctx.error など) の場合、レスポンスは以下のようになります。

```
{
    "data": {
        "getPost": {
            "id": "1",
            "title: "A post"
        }
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "ConditionalCheckFailedException",
            "message": "Conditional check failed exception...",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

2017-02-28 から 2018-05-29 への移行

2017-02-28 から 2018-05-29 への移行は簡単です。リゾルバーのリクエストマッピングテンプレートの version フィールドまたは関数の version オブジェクトを変更します。ただし、2018-05-29 と 2017-02-28 では実行動作が異なるため、変更点についてこちらで概要を説明しています。

2017-02-28 から 2018-05-29 への同じ実行動作の保持

場合によっては、同じ実行動作を保持できる場合もあります。tたとえば2018-05-29バージョン管理されたテンプレートを実行しながら2017-02-28を実行することが可能です。

例: DynamoDB PutItem

以下の 2017-02-28 DynamoDB PutItem リクエストテンプレートがあるとします。

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
```

```
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "attributeValues" : {
        "baz" : ... typed value
    },
    "condition" : {
       ...
    }
}
```

また、以下のレスポンステンプレートがあるとします。

```
$util.toJson($ctx.result)
```

2018-05-29 に移行すると、これらのテンプレートは以下のように変更されます。

```
{
    "version" : "2018-05-29", ## Note the new 2018-05-29 version
    "operation" : "PutItem",
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "attributeValues" : {
        "baz" : ... typed value
    },
    "condition" : {
       ...
    }
}
```

また、レスポンステンプレートは以下のように変更されます。

```
## If there is a datasource invocation error, we choose to raise the same error
## the field data will be set to null.
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end

## If the data source invocation is null, we return null.
```

```
#if($util.isNull($ctx.result))
  #return
#end

$util.toJson($ctx.result)
```

エラーを処理するのはお客様であるため、DynamoDB から返された `$util.error()` を使用して同じエラーが発生するようにしました。マッピングテンプレートを 2018-05-29 に変換するようにこのスニペットを調整できます。レスポンステンプレートが異なる場合は、実行動作が変わることを考慮する必要があります。

例: DynamoDB GetItem

以下の 2017-02-28 DynamoDB GetItem リクエストテンプレートがあるとします。

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true
}
```

また、以下のレスポンステンプレートがあるとします。

```
## map table attribute postId to field Post.id
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))

$util.toJson($ctx.result)
```

2018-05-29 に移行すると、これらのテンプレートは以下のように変更されます。

```
{
    "version" : "2018-05-29", ## Note the new 2018-05-29 version
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
```

```
    "consistentRead" : true
}
```

また、レスポンステンプレートは以下のように変更されます。

```
## If there is a datasource invocation error, we choose to raise the same error
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type)
#end

## If the data source invocation is null, we return null.
#if($util.isNull($ctx.result))
  #return
#end

## map table attribute postId to field Post.id
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))

$util.toJson($ctx.result)
```

2017-02-28 バージョンでは、データソースの呼び出しが null の場合は、DynamoDB テーブルにキーと一致する項目がないことを意味します。そのため、レスポンスマッピングテンプレートは実行されません。ほとんどの場合は問題ありませんが、`$ctx.result` が null ではないと想定される場合は、そのシナリオを処理する必要があります。

2017-02-28

特性

- データソースの呼び出し結果が null の場合、レスポンスマッピングテンプレートは実行されません。
- データソースの呼び出しでエラーが発生した場合、レスポンスマッピングテンプレートは実行され、評価結果は GraphQL レスポンスの `errors.data` ブロック内に挿入されます。

GraphQL タイプのリファレンス

GraphQL のスカラー型は、GraphQL スキーマのプリミティブリーフ値を表します。これらは、単一の値を解決する最も基本的なデータ型です。オブジェクトタイプとは異なり、スカラー型にサブフィールドを含めることはできません。GraphQL にはデフォルトのスカラー型のセットが付属しています。

- Int: 符号付き 32 ビット整数
- Float: 符号付き倍精度浮動小数点値
- String: UTF-8 文字シーケンス
- Boolean: true または false の値
- ID: オブジェクトの再取得やキャッシュのキーとしてよく使用される一意の識別子

これらのスカラー型は、スキーマ内のより複雑なタイプの構成要素として機能します。これらは、シンプルで単一の値を含むフィールドを定義するために使用されます。これらの組み込みスカラーに加えて、 はさまざまなユースケース用に追加のスカラー AWS AppSync を提供します。

GraphQL の Interface と Union は、柔軟で拡張可能なスキーマ設計を可能にする抽象型です。これらは、関連するタイプをグループ化し、多相型クエリを有効にするメカニズムを提供します。GraphQL の Interface は、Interface を実装するためにタイプに含める必要があるフィールドのセットを定義する抽象型です。これは、実装タイプが持つ必要があるフィールドの共通セットを指定することで、オブジェクトの契約として機能します。Interface は、いくつかのフィールドを保証しながら、複数の異なる型を持つオブジェクトまたはフィールドを返す場合に便利です。対照的に、GraphQL の Union は、いくつかのオブジェクトタイプのいずれかになる可能性のあるタイプを表しますが、それらのタイプ間で一般的なフィールドは定義しません。Union は、複数のタイプのフィールドを返す必要がある場合に役立ちます。これらのタイプは、必ずしも共通のフィールドを共有しているわけではありません。Interface と Union の両方は、フィールドが異なるタイプのデータを返す可能性があるシナリオで特に有用であり、クライアントは返されたタイプに基づいて特定のフィールドをクエリできます。

このセクションは、スキーマタイプのリファレンスとして使用されます。

トピック

- GraphQL のスカラー型
- GraphQL の Interface と Union

GraphQL のスカラー型

GraphQL オブジェクトタイプには名前とフィールドがあり、これらのフィールドにはサブフィールドを使用できます。最終的には、オブジェクトタイプのフィールドを、クエリの各要素を表すスカラー型に解決する必要があります。オブジェクトタイプとスカラーの詳細については、GraphQL ウェブサイトの「スキーマとタイプ」を参照してください。

デフォルトの GraphQL スカラーセットに加えて、 では、 AWS プレフィックスで始まるサービス定義スカラーを使用 AWS AppSync することもできます。 AWS AppSync では、ユーザー定義 (カスタム) スカラーの作成はサポートされていません。デフォルトまたは AWS スカラーのいずれかを使用する必要があります。

カスタムオブジェクトタイプのプレフィックスとして AWS を使用することはできません。

次のセクションはスキーマタイピングのリファレンスです。

デフォルトスカラー

GraphQL は、次のデフォルトスカラー を定義します。

デフォルトのスカラーのリスト

`ID`

オブジェクトの一意な識別子。このスカラーは、`String` のようにシリアル化されますが、人間が読めることは意図していません。

`String`

UTF-8 文字シーケンス。

`Int`

$-(2^{31})$ と $2^{31}-1$の間の整数値。

`Float`

IEEE 754 浮動小数点値

`Boolean`

ブール値 (`true` または `false`)。

AWS AppSync スカラー

AWS AppSync は、次のスカラーを定義します。

AWS AppSync スカラーリスト

`AWSDate`

拡張機能ISO 8601 の日付形式の文字列YYYY-MM-DD。

`AWSTime`

拡張機能ISO 8601形式の文字列hh:mm:ss.sss。

`AWSDateTime`

拡張機能ISO 8601 の日時番号形式の文字列YYYY-MM-DDThh:mm:ss.sssZ。

> **Note**
>
> AWSDate、AWSTime、および AWSDateTime スカラーは、必要に応じてタイムゾーンオフセットに含むことができます。例えば、値 1970-01-01Z、1970-01-01-07:00、 および 1970-01-01+05:30 はすべて AWSDate に有効です。タイムゾーンのオフセットは、Z(UTC)、または時間と分 (およびオプションで秒) のオフセットのいずれかである必要があります。例えば、±hh:mm:ss と指定します。ISO 8601 標準には含まれていませんが、タイムゾーンオフセットの第 2 フィールドは有効と見なされます。

`AWSTimestamp`

1970-01-01-T00:00Z 前後の秒数を表す整数値。

`AWSEmail`

RFC 822で定義される local-part@domain-part 形式のメールアドレス。

`AWSJSON`

JSON 文字列。すべての有効な JSON コンストラクトは、リテラルな入力文字列としてではなく、マップ、リスト、スカラー値として自動的に解析され、リゾルバーマッピングテンプレートにロードされます。引用符で囲まれていない文字列、または無効な JSON は GraphQL 検証エラーになります。

AWSPhone

電話番号 この値は文字列として保存されます。電話番号には、スペースまたはハイフンのいずれかを指定して、数字グループを区切ることができます。国番号のない電話番号は、北米番号計画 (NANP)に紐づいている米国/北米の電話番号とみなされます。

AWSURL

RFC 1738によって定義される URL。例えば、https://www.amazon.com/dp/B000NZW3KC/、mailto:example@example.com です。URL にはスキーマ (http,mailto) および 2 つのフォワードスラッシュ (//) をパス部分に入れる必要があります。

AWSIPAddress

有効な IPv4 または IPv6 アドレス。IPv4 アドレスはクアッドドット表記で想定されます (123.12.34.56)。IPv6 アドレスは、角カッコで囲まれていないコロン区切りの形式で想定されます (1a2b:3c4b::1234:4567)。オプションの CIDR サフィックス (123.45.67.89/16) を含めることで、サブネットマスクを示すことができます。

スキーマの使用例

以下に示す GraphQL スキーマの例は、すべてのカスタムのスカラーを「オブジェクト」として使用するとともに、基本的なput、get、および list 操作にリゾルバーリクエストとレスポンステンプレートを使用しています。最後に、 クエリとミューテーションを実行するときに、この をどのように 使用できるかの例を示します。

```
type Mutation {
    putObject(
        email: AWSEmail,
        json: AWSJSON,
        date: AWSDate,
        time: AWSTime,
        datetime: AWSDateTime,
        timestamp: AWSTimestamp,
        url: AWSURL,
        phoneno: AWSPhone,
        ip: AWSIPAddress
    ): Object
}

type Object {
    id: ID!
```

```
    email: AWSEmail
    json: AWSJSON
    date: AWSDate
    time: AWSTime
    datetime: AWSDateTime
    timestamp: AWSTimestamp
    url: AWSURL
    phoneno: AWSPhone
    ip: AWSIPAddress
}

type Query {
    getObject(id: ID!): Object
    listObjects: [Object]
}

schema {
    query: Query
    mutation: Mutation
}
```

putObject のリクエストテンプレートは次のようになります。putObject は PutItem オペレーションを使用して Amazon DynamoDB テーブル内の項目を作成または更新します。このコードスニペットには、データソースとして Amazon DynamoDB テーブルが設定されていないことに注意してください。これは例としてのみ使用されています。

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

putObject のレスポンステンプレートは結果を返します。

```
$util.toJson($ctx.result)
```

getObject のリクエストテンプレートは次のようになります。getObject は GetItem オペレーションを開始し、指定されたプライマリキーを持つ項目の属性のセットを取得します。このコードス

二ペットには、データソースとして Amazon DynamoDB テーブルが設定されていないことに注意してください。これは例としてのみ使用されています。

```
{
    "version": "2017-02-28",
    "operation": "GetItem",
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
    }
}
```

getObject のレスポンステンプレートは結果を返します。

```
$util.toJson($ctx.result)
```

listObjects のリクエストテンプレートは次のようになります。listObjects は Scan オペレーションを使用して 1 つ以上の項目と属性を返します。このコードス二ペットには、データソースとして Amazon DynamoDB テーブルが設定されていないことに注意してください。これは例としてのみ使用されています。

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
}
```

listObjects のレスポンステンプレートは結果を返します。

```
$util.toJson($ctx.result.items)
```

このスキーマを GraphQL クエリと共に使用する例をいくつか次に示します。

```
mutation CreateObject {
    putObject(email: "example@example.com"
        json: "{\"a\":1, \"b\":3, \"string\": 234}"
        date: "1970-01-01Z"
        time: "12:00:34."
        datetime: "1930-01-01T16:00:00-07:00"
        timestamp: -123123
        url:"https://amazon.com"
        phoneno: "+1 555 764 4377"
```

```
        ip: "127.0.0.1/8"
    ) {
        id
        email
        json
        date
        time
        datetime
        url
        timestamp
        phoneno
        ip
    }
}

query getObject {
    getObject(id:"0d97daf0-48e6-4ffc-8d48-0537e8a843d2"){
        email
        url
        timestamp
        phoneno
        ip
    }
}

query listObjects {
    listObjects {
        json
        date
        time
        datetime
    }
}
```

GraphQL の Interface と Union

GraphQL 型システムは Interface をサポートしています。インターフェイスを実装するためにタイプに含める必要がある、フィールドの特定セットをインターフェイスで公開します。

GraphQL 型システムはUnionもサポートしています。Union はインターフェイスと同じです。異なるのはフィールドの共通セットを定義しない点です。Union は、可能なタイプが論理階層を共有しないとき、一般的にインターフェイスより優先されます。

次のセクションは、スキーマの型付けに関するリファレンスです。

インターフェースの例

たとえば、何らかのアクティビティまたは人の集まりを Event インターフェイスで表すことができます。考えられるイベントタイプにはConcert、Conference、Festival などがあります。このようなタイプにはすべて、共通の特性 (名前、イベントの開催場所、開始日および終了日など) があります。これらのタイプに差分もあります。Conference には、講演者やワークショップのリストがあり、Concert には演奏するバンドが記載されています。

スキーマ定義言語 (SDL) では、Event インターフェイスは次のように定義されます。

```
interface Event {
        id: ID!
        name : String!
        startsAt: String
        endsAt: String
        venue: Venue
        minAgeRestriction: Int
}
```

また、Event インターフェイスを実装する各タイプは次のようになります。

```
type Concert implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performingBand: String
}

type Festival implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performers: [String]
}
```

```
type Conference implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    speakers: [String]
    workshops: [String]
}
```

インターフェイスは数種類の要素がある場合を表すのに便利です。たとえば、特定の会場で開催されているすべてのイベントを検索できます。次のように findEventsByVenue フィールドをスキーマに追加します。

```
schema {
    query: Query
}

type Query {
    # Retrieve Events at a specific Venue
    findEventsAtVenue(venueId: ID!): [Event]
}

type Venue {
    id: ID!
    name: String
    address: String
    maxOccupancy: Int
}

type Concert implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performingBand: String
}

interface Event {
```

```
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
}

type Festival implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performers: [String]
}

type Conference implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    speakers: [String]
    workshops: [String]
}
```

findEventsByVenue は Event のリストを返します。GraphQL インターフェイスフィールドはすべての実装タイプに共通しているため、Event インターフェイス (id、name、startsAt、endsAt、venue、および minAgeRestriction) の任意のフィールドを選択できます。さらに、GraphQL の フラグメントを使用して、タイプを指定している限り、任意の実装タイプのフィールドにアクセスできます。

インターフェイスを使用する GraphQL クエリの例を詳しく見てみましょう。

```
query {
  findEventsAtVenue(venueId: "Madison Square Garden") {
    id
    name
    minAgeRestriction
    startsAt
```

```
    ... on Festival {
      performers
    }

    ... on Concert {
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

上記のクエリは、結果の単一のリストを生み出し、サーバーで、デフォルトでは、開始日によりイベントをソートできます。

```
{
  "data": {
    "findEventsAtVenue": [
      {
        "id": "Festival-2",
        "name": "Festival 2",
        "minAgeRestriction": 21,
        "startsAt": "2018-10-05T14:48:00.000Z",
        "performers": [
          "The Singers",
          "The Screamers"
        ]
      },
      {
        "id": "Concert-3",
        "name": "Concert 3",
        "minAgeRestriction": 18,
        "startsAt": "2018-10-07T14:48:00.000Z",
        "performingBand": "The Jumpers"
      },
      {
        "id": "Conference-4",
        "name": "Conference 4",
        "minAgeRestriction": null,
```

```
        "startsAt": "2018-10-09T14:48:00.000Z",
        "speakers": [
          "The Storytellers"
        ],
        "workshops": [
          "Writing",
          "Reading"
        ]
      }
    ]
  }
}
```

結果はイベントの 1 つのコレクションとして返されるため、共通特性を表すためにインターフェイスを使用すると、結果のソートに大変役立ちます。

ユニオンの例

先に述べたように、ユニオンは共通のフィールドセットを定義しません。検索結果で多様なタイプを表すことがあります。Event スキーマを使用して、次のように SearchResult ユニオンを定義できます。

```
type Query {
    # Retrieve Events at a specific Venue
    findEventsAtVenue(venueId: ID!): [Event]
    # Search across all content
    search(query: String!): [SearchResult]
}

union SearchResult = Conference | Festival | Concert | Venue
```

この場合、SearchResult ユニオンで任意のフィールドをクエリするには、フラグメントを使用する必要があります。

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }
```

```
    ... on Festival {
      id
      name
      performers
    }

    ... on Concert {
      id
      name
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

AWS AppSync でのタイプ解決

タイプ解決は、GraphQL エンジンが特定のオブジェクトタイプとして解決された値を識別するメカニズムです。

ユニオン検索例に戻り、クエリで結果を出した場合、結果リストの各項目は、定義された `SearchResult` ユニオンの可能なタイプの 1 つ (つまり、`Conference`、`Festival`、`Concert`、または `Venue`) として、それ自体を示す必要があります。

`Festival` を `Venue` や `Conference` から識別するロジックは、アプリケーションの要件に依存するため、GraphQL エンジンには、そのままの結果から可能なタイプを識別するヒントを与える必要があります。

AWS AppSync では、このヒントは という名前のメタフィールドで表されます。このメタフィールドの値は__typename、識別されたオブジェクトタイプ名に対応します。 __typenameは、インターフェイスまたはユニオンである戻り型に必要です。

タイプ解決の例

前のスキーマを再利用します。コンソールに移動して、[スキーマ] ページで以下を追加することで、手順どおりに進めることができます。

```
schema {
```

```
    query: Query
}

type Query {
    # Retrieve Events at a specific Venue
    findEventsAtVenue(venueId: ID!): [Event]
    # Search across all content
    search(query: String!): [SearchResult]
}

union SearchResult = Conference | Festival | Concert | Venue

type Venue {
    id: ID!
    name: String!
    address: String
    maxOccupancy: Int
}

interface Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
}

type Festival implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performers: [String]
}

type Conference implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
```

```
    minAgeRestriction: Int
    speakers: [String]
    workshops: [String]
}

type Concert implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performingBand: String
}
```

リゾルバーを Query.search フィールドにアタッチしましょう。Resolvers セクションで、[アタッチ] を選択し、NONE タイプの新しいデータソースを作成してから、StubDataSource という名前を付けます。この例では、外部ソースから結果を取得したものとし、リクエストマッピングテンプレートで取得された結果をハードコードします。

リクエストマッピングテンプレートペインで、次のように入力します。

```
{
    "version" : "2018-05-29",
    "payload":
    ## We are effectively mocking our search results for this example
    [
        {
            "id": "Venue-1",
            "name": "Venue 1",
            "address": "2121 7th Ave, Seattle, WA 98121",
            "maxOccupancy": 1000
        },
        {
            "id": "Festival-2",
            "name": "Festival 2",
            "performers": ["The Singers", "The Screamers"]
        },
        {
            "id": "Concert-3",
            "name": "Concert 3",
            "performingBand": "The Jumpers"
        },
```

```
        {
            "id": "Conference-4",
            "name": "Conference 4",
            "speakers": ["The Storytellers"],
            "workshops": ["Writing", "Reading"]
        }
    ]
}
```

アプリケーションが、id フィールドの一部としてタイプ名を返す場合、タイプ解決ロジックは、id フィールドを解析してタイプ名を抽出し、__typename フィールドを各結果を追加します。次のようにレスポンスマッピングテンプレートでそのロジックを実行できます。

> **Note**
>
> Lambda データソースを使用している場合は、このタスクを Lambda 関数の一部として実行することもできます。

```
#foreach ($result in $context.result)
    ## Extract type name from the id field.
    #set( $typeName = $result.id.split("-")[0] )
    #set( $ignore = $result.put("__typename", $typeName))
#end
$util.toJson($context.result)
```

次のクエリを実行します。

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
        id
      name
      performers
    }
```

```
    ... on Concert {
      id
      name
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

クエリにより、次の結果が返されます。

```
{
  "data": {
    "search": [
      {
        "id": "Venue-1",
        "name": "Venue 1",
        "address": "2121 7th Ave, Seattle, WA 98121"
      },
      {
        "id": "Festival-2",
        "name": "Festival 2",
        "performers": [
          "The Singers",
          "The Screamers"
        ]
      },
      {
        "id": "Concert-3",
        "name": "Concert 3",
        "performingBand": "The Jumpers"
      },
      {
        "speakers": [
          "The Storytellers"
        ],
        "workshops": [
          "Writing",
```

```
          "Reading"
        ]
      }
    ]
  }
}
```

タイプ解決ロジックはアプリケーションによって異なります。たとえば、特定のフィールドまたはフィールドの組み合わせの存在をチェックする、異なる識別ロジックがあってもかまいません。つまり、performers フィールドの存在を検出して、Festival を識別する、または speakers と workshops フィールドの組み合わせを検出して、Conference を識別できます。最終的に、使用するロジックを定義するのはユーザーです。

のトラブルシューティングと一般的なミス AWS AppSync

このセクションでは、いくつかの一般的なエラーとそのトラブルシューティング方法について説明します。

DynamoDB キーのマッピングが正しくない

GraphQL 処理が次のエラーメッセージを返した場合、リクエストマッピングテンプレートの構造が Amazon DynamoDB のキーの構造と一致していない可能性があります。

```
The provided key element does not match the schema (Service: AmazonDynamoDBv2; Status
 Code: 400; Error Code
```

たとえば、DynamoDB テーブルに "id" というハッシュキーがあり、テンプレートが次の例のように "PostID" を指示している場合、"id" は "PostID" と一致しないため上記のようなエラーになります。

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "PostID" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

リゾルバーがない

クエリといった GraphQL 処理を実行して null レスポンスを受信した場合は、リゾルバーを設定していない可能性があります。

たとえば、getCustomer(userId: ID!): フィールドを定義するスキーマをインポートし、このフィールドにリゾルバーを設定しなかった場合、getCustomer(userId:"ID123"){...} などのクエリを実行すると、次のようなレスポンスを受信します。

```
{
    "data": {
    "getCustomer": null
    }
```

```
}
```

マッピングテンプレートのエラー

マッピングテンプレートが正しく設定されていない場合、errorType が MappingTemplate の GraphQL レスポンスを受信します。message フィールドには、マッピングテンプレートに問題があることが示されます。

たとえば、リクエストマッピングテンプレートに operation フィールドがない場合、または operation フィールド名が正しくない場合、次のようなレスポンスを受信します。

```
{
    "data": {
        "searchPosts": null
    },
    "errors": [
        {
        "path": [
            "searchPosts"
        ],
        "errorType": "MappingTemplate",
        "locations": [
            {
            "line": 2,
            "column": 3
            }
        ],
        "message": "Value for field '$[operation]' not found."
        }
    ]
}
```

戻り値の型が正しくない

データソースからの戻り値の型は、スキーマに定義したオブジェクトの型と一致している必要があります。一致しない場合、次のような GraphQL エラーが表示されます。

```
"errors": [
    {
    "path": [
```

```
        "posts"
    ],
    "locations": null,
    "message": "Can't resolve value (/posts) : type mismatch error, expected type LIST,
 got OBJECT"
    }
]
```

このような状況は、たとえば次のようなクエリ定義により発生します。

```
type Query {
    posts: [Post]
}
```

これは [Posts] オブジェクトのリストを期待します。たとえば次のように、Node.JS に Lambda 関数があるとします。

```
const result = { data: data.Items.map(item => { return item ; }) };
callback(err, result);
```

result はオブジェクトであるため、エラーがスローされます。result.data へのコールバックを変更するか、リストを返さないようにスキーマを変更する必要があります。

無効なリクエストの処理

AWS AppSync が (無効な構文などの不適切なデータが原因で) フィールドリゾルバーにリクエストを処理および送信できない場合、レスポンスペイロードは に設定された値nullと関連するエラーを含むフィールドデータを返します。

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。