Developer Guide

AWS AppSync GraphQL

AWS AppSync GraphQL: Developer Guide

Table of Contents

What is AWS AppSync?

AWS AppSync enables developers to connect their applications and services to data and events with secure, serverless and high-performing GraphQL and Pub/Sub APIs. You can do the following with AWS AppSync:

- Access data from one or more data sources from a single GraphQL API endpoint.
- Combine multiple source GraphQL APIs into a single, merged GraphQL API.
- Publish real-time data updates to your applications.
- Leverage built-in security, monitoring, logging, and tracing, with optional caching for low latency.
- Only pay for API requests and any real-time messages that are delivered.

Important

As of Mar 13, 2025, you can build a real-time PubSub API powered by WebSockets using AWS AppSync Events. For more information, see Publish events via WebSocket in the *AWS AppSync Events Developer Guide*.

Topics

AWS AppSync features

- Simplified data access and querying, powered by GraphQL
- Serverless WebSockets for GraphQL subscriptions and pub/sub channels
- Server-side caching to make data available in high speed in-memory caches for low latency
- JavaScript and TypeScript support to write business logic
- Enterprise security with Private APIs to restrict API access and integration with AWS WAF

- Built in authorization controls, with support for API keys, IAM, Amazon Cognito, OpenID Connect providers, and Lambda authorization for custom logic.
- Merged APIs to support federated use cases

For more details about each of these capabilities, see AWS AppSync features.

Are you a first-time AWS AppSync user?

We recommend that first-time AWS AppSync users begin by reading the following sections:

- If you're unfamiliar with GraphQL, see the *Getting started: Creating your first GraphQL API*.
- If you're building applications that consume GraphQL APIs, see *Building a client application* and the section called "Using subscriptions for real-time data applications".
- If you're looking for GraphQL resolver information, see the following:

  JavaScript/TypeScript

  - Resolver tutorials (JavaScript)
  - Resolver reference (JavaScript)

  VTL

  - Resolver tutorials (VTL)
  - *Resolver mapping template reference (VTL)*
- If you're looking for AWS AppSync example projects, updates, and more, see the AppSync blog.

Related services

If you're building a web or mobile app from the ground up, consider using AWS Amplify. Amplify leverages AWS AppSync and other AWS services to help you build more robust, powerful web and mobile apps with less work.

Pricing for AWS AppSync

AWS AppSync is priced based on millions of requests and updates. Caching costs an additional fee. For more information, see AWS AppSync pricing.

The following lists the exceptions to general AWS AppSync pricing:

- Requests are not charged for authorization and authentication failures.
- Calls to methods that require API keys are not charged when API keys are missing or invalid.

GraphQL and AWS AppSync architecture

> **Note**
>
> This guide assumes the user has a working knowledge of the REST architectural style. We recommend reviewing this and other front-end topics before working with GraphQL and AWS AppSync.

GraphQL is a query and manipulation language for APIs. GraphQL provides a flexible and intuitive syntax to describe data requirements and interactions. It enables developers to ask for exactly what is needed and get back predictable results. It also makes it possible to access many sources in a single request, reducing the number of network calls and bandwidth requirements, therefore saving battery life and CPU cycles consumed by applications.

Making updates to data is made simple with mutations, allowing developers to describe how the data should change. GraphQL also facilitates the quick setup of real-time solutions via subscriptions. All of these features combined, coupled with powerful developer tools, make GraphQL essential to managing application data.

GraphQL is an alternative to REST. RESTful architecture is currently one of the more popular solutions for client-server communication. It's centered on the concept of your resources (data) being exposed by a URL. These URLs can be used to access and manipulate the data through CRUD (create, read, update, delete) operations in the form of HTTP methods like `GET`, `POST`, and `DELETE`. REST's advantage is that it's relatively simple to learn and implement. You can quickly set up RESTful APIs to call a wide range of services.

However, technology is getting more complicated. As applications, tools, and services begin to scale for a worldwide audience, the need for fast, scalable architectures is of paramount importance. REST has many shortcomings when dealing with scalable operations. See this use case for an example.

In the following sections, we'll review some of the concepts surrounding RESTful APIs. We'll then introduce GraphQL and how it works.

For more information about GraphQL and the benefits of migrating over to AWS, see the Decision guide to GraphQL implementations.

Topics

What is an API?

An application programming interface (API) defines the rules that you must follow to communicate with other software systems. Developers expose or create APIs so that other applications can communicate with their applications programmatically. For example, the timesheet application exposes an API that asks for an employee's full name and a range of dates. When it receives this information, it internally processes the employee's timesheet and returns the number of hours worked in that date range.

You can think of a web API as a gateway between clients and resources on the web.

Clients

Clients are users who want to access information from the web. The client can be a person or a software system that uses the API. For example, developers can write programs that access weather data from a weather system. Or you can access the same data from your browser when you visit the weather website directly.

Resources

Resources are the information that different applications provide to their clients. Resources can be images, videos, text, numbers, or any type of data. The machine that gives the resource to the client is also called the server. Organizations use APIs to share resources and provide web services while maintaining security, control, and authentication. In addition, APIs help them to determine which clients get access to specific internal resources.

What is REST?

At a high level, representational State Transfer (REST) is a software architecture that imposes conditions on how an API should work. REST was initially created as a guideline to manage communication on a complex network like the internet. You can use REST-based architecture to support high-performing and reliable communication at scale. You can easily implement and modify it, bringing visibility and cross-platform portability to any API system.

API developers can design APIs using several different architectures. APIs that follow the REST architectural style are called REST APIs. Web services that implement REST architecture are called RESTful web services. The term RESTful API generally refers to RESTful web APIs. However, you can use the terms REST API and RESTful API interchangeably.

The following are some of the principles of the REST architectural style:

Uniform interface

The uniform interface is fundamental to the design of any RESTful webservice. It indicates that the server transfers information in a standard format. The formatted resource is called a representation in REST. This format can be different from the internal representation of the resource on the server application. For example, the server can store data as text but send it in an HTML representation format.

Uniform interface imposes four architectural constraints:

1. Requests should identify resources. They do so by using a uniform resource identifier.
2. Clients have enough information in the resource representation to modify or delete the resource if they want to. The server meets this condition by sending metadata that describes the resource further.
3. Clients receive information about how to process the representation further. The server achieves this by sending self-descriptive messages that contain metadata about how the client can best use them.
4. Clients receive information about all other related resources they need to complete a task. The server achieves this by sending hyperlinks in the representation so that clients can dynamically discover more resources.

Statelessness

In REST architecture, statelessness refers to a communication method in which the server completes every client request independently of all previous requests. Clients can request resources in any order, and every request is stateless or isolated from other requests. This REST API design constraint implies that the server can completely understand and fulfill the request every time.

Layered system

In a layered system architecture, the client can connect to other authorized intermediaries between the client and server, and it will still receive responses from the server. Servers can also pass on requests to other servers. You can design your RESTful web service to run on several servers with multiple layers such as security, application, and business logic, working together to fulfill client requests. These layers remain invisible to the client.

Cacheability

RESTful web services support caching, which is the process of storing some responses on the client or on an intermediary to improve server response time. For example, suppose that you visit a website that has common header and footer images on every page. Every time you visit a new website page, the server must resend the same images. To avoid this, the client caches or stores these images after the first response and then uses the images directly from the cache. RESTful web services control caching by using API responses that define themselves as cacheable or noncacheable.

What is a RESTful API?

RESTful API is an interface that two computer systems use to exchange information securely over the internet. Most business applications have to communicate with other internal and third-party applications to perform various tasks. For example, to generate monthly payslips, your internal accounts system has to share data with your customer's banking system to automate invoicing and communicate with an internal timesheet application. RESTful APIs support this information exchange because they follow secure, reliable, and efficient software communication standards.

How do RESTful APIs work?

The basic function of a RESTful API is the same as browsing the internet. The client contacts the server by using the API when it requires a resource. API developers explain how the client should

use the REST API in the server application API documentation. These are the general steps for any REST API call:

1. The client sends a request to the server. The client follows the API documentation to format the request in a way that the server understands.
2. The server authenticates the client and confirms that the client has the right to make that request.
3. The server receives the request and processes it internally.
4. The server returns a response to the client. The response contains information that tells the client whether the request was successful. The response also includes any information that the client requested.

The REST API request and response details vary slightly depending on how the API developers design the API.

What is GraphQL?

GraphQL is both a query language for APIs and a runtime for executing those queries. GraphQL allows clients to request exactly the data they need, providing a more flexible and efficient alternative to REST in many scenarios. Unlike REST, which relies on predefined endpoints, GraphQL uses a single endpoint where clients can specify their data requirements in the form of queries and mutations.

See Components of a GraphQL API for more information on how GraphQL APIs are structured.

Comparing REST and GraphQL

APIs (Application Programming Interfaces) play a crucial role in facilitating data exchange between applications. As stated earlier, two prominent approaches for designing APIs have emerged: GraphQL and REST. While both serve the fundamental purpose of enabling client-server communication, they differ significantly in their implementation and use cases.

GraphQL and REST share several key characteristics:

1. **Client-Server Model**: Both use a client-server architecture for data exchange.
2. **Statelessness**: Neither maintains client session information between requests.

3. **HTTP-Based**: Both typically use HTTP as the underlying communication protocol.
4. **Resource-Oriented Design**: Both design their data interchange around resources, which refer to any data or object that the client can access and manipulate through the API.
5. **Data Format Flexibility**: JSON is the most commonly used data exchange format in both, though other formats like XML and HTML are also supported.
6. **Language and Database Agnostic**: Both can work with any programming language or database structure, making them highly interoperable.
7. **Caching Support**: Both support caching, allowing clients and servers to store frequently accessed data for improved performance.

While sharing some fundamental principles, GraphQL and REST differ significantly in their approach to API design and data fetching:

1. **Request Structure and Data Fetching**

   REST uses different HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources. This often requires multiple endpoints for different resources, which can lead to inefficiencies in data retrieval. For example, running a GET operation to retrieve a user's data may lead to data over-fetching or under-fetching. To get the correct data, truncation or multiple operations may be called.

   GraphQL uses a single endpoint for all operations. It relies on queries for fetching data and mutations for modifying data. Clients can use queries to fetch exactly the data they need in a single request, which reduces network overhead by minimizing data transfer.

2. **Server-side Schema**

   REST doesn't require a server-side schema, though one can be optionally defined for efficient API design and documentation.

   GraphQL uses a strongly-typed server-side schema to define data and data services. The schema, written in GraphQL Schema Definition Language (SDL), includes object types and fields for each object and server-side resolver functions that define operations for each field.

3. **Versioning**

   REST often includes versioning in the URL, which can lead to maintaining multiple API versions simultaneously. Versioning is not mandatory but can help prevent breaking changes.

GraphQL promotes a continuous evolution of the API without explicit versioning by requiring backward compatibility. Deleted fields return error messages, while deprecation tags phase out old fields and return warning messages.

4. **Error Handling**

   REST is weakly typed, requiring error handling to be built into the surrounding code. This may not automatically identify type-related errors (e.g., parsing a number as text).

   By contrast, GraphQL is strongly typed and requires a comprehensive schema definition. This allows your service to automatically identify many request errors with a high level of detail.

5. **Use Cases**

   REST is better suited for:

   - Smaller applications with less complex data requirements.
   - Scenarios where data and operations are used similarly by all clients.
   - Applications without complex data querying needs.

   GraphQL is better suited for:

   - Scenarios with limited bandwidth, where minimizing requests and responses is crucial.
   - Applications with multiple data sources that need to be combined at a single endpoint.
   - Cases where client requests vary significantly and expect different response structures.

   Note that it's possible to use both GraphQL and REST APIs within a single application for different areas of functionality. Furthermore, you can upgrade a RESTful API to include GraphQL capabilities without a complete rewrite. See How to build GraphQL resolvers for AWS data sources for an example.

Why Use GraphQL over REST?

REST is one of the cornerstone architectural styles of web APIs. However, as the world becomes more interconnected, the need to develop robust and scalable applications will become a more pressing issue. While REST is often used to build web APIs, there are several recurring drawbacks to RESTful implementations that have been identified:

1. **Data requests**: Using RESTful APIs, you would typically request the data you need through endpoints. The problem arises when you have data that may not be so neatly packaged. The

data you need may be sitting behind multiple layers of abstraction, and the only way to fetch the data is by using multiple endpoints, which means making multiple requests to extract all of the data.

2. **Overfetching and underfetching**: To add to the problems of multiple requests, the data from each endpoint is strictly defined, meaning you will return whatever data was defined for that API, even if you didn't technically want it.

   This can result in *over-fetching*, which means our requests return superfluous data. For example, let's say you're requesting company personnel data and want to know the names of the employees in a certain department. The endpoint that returns the data will contain the names, but it might also contain other data like job title or date of birth. Because the API is fixed, you can't just request the names alone; the rest of the data comes with it.

   The opposite situation in which we don't return enough data is called *under-fetching*. To get all of the requested data, you may have to make multiple requests to the service. Depending on how the data was structured, you could run into inefficient queries resulting in issues like the dreaded n+1 problem.

3. **Slow development iterations**: Many developers tailor their RESTful APIs to fit the flow of their applications. However, as their applications grow, both the front- and backends may require extensive changes. As a result, the APIs may no longer fit the shape of the data in a way that's efficient or impactful. This results in slower product iterations due to the need for API modifications.

4. **Performance at scale**: Due to these compounding issues, there are many areas where scalability will be impacted. Performance on the application side may be impacted because your requests will return too much data or too little (resulting in more requests). Both situations cause unnecessary strain on the network resulting in poor performance. On the developer side, the speed of development may be reduced because your APIs are fixed and no longer fit the data they're requesting.

GraphQL's selling point is to overcome the drawbacks of REST. Here are some of the key solutions GraphQL offers to developers:

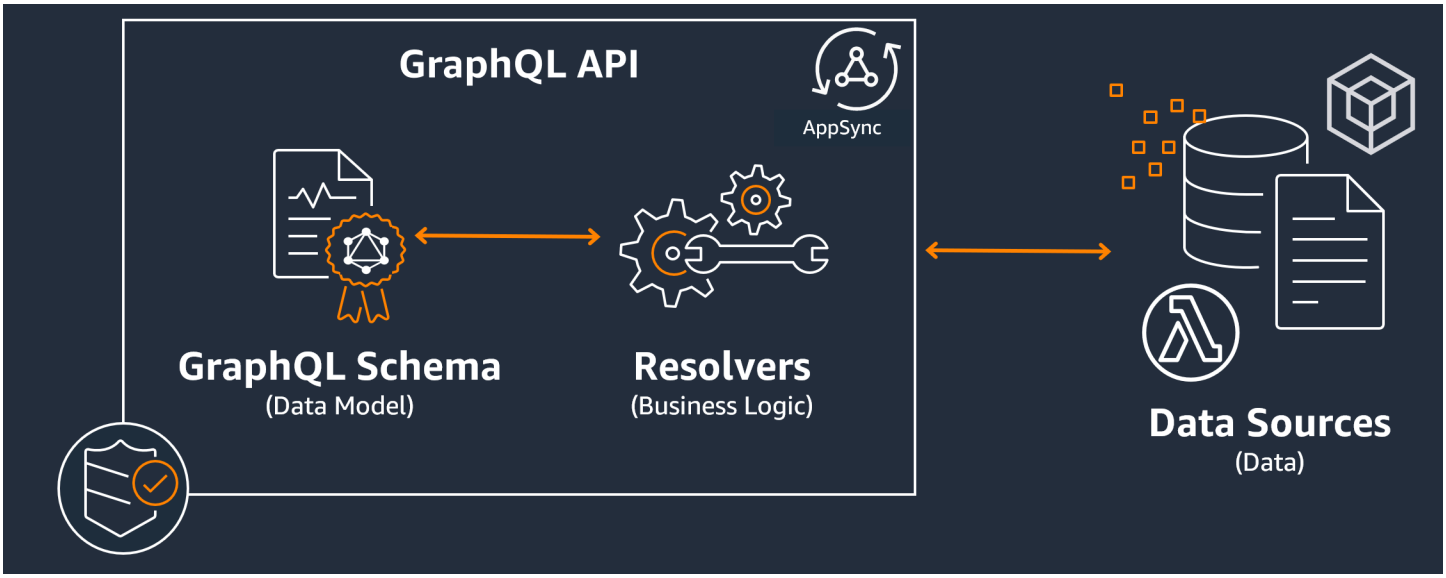
1. **Single endpoints**: GraphQL uses a single endpoint to query data. There's no need to build multiple APIs to fit the shape of the data. This results in fewer requests going over the network.

2. **Fetching**: GraphQL solves the perennial issues of over- and under-fetching by simply defining the data you need. GraphQL lets you shape the data to fit your needs so you only receive what you asked for.
3. **Abstraction**: GraphQL APIs contain a few components and systems that describe the data using a language-agnostic standard. In other words, the shape and structure of the data are standardized so both the front- and backends know how it will be sent over the network. This allows developers on both ends to work with GraphQL's systems and not around them.
4. **Rapid iterations**: Because of the standardization of data, changes on one end of development may not be required on the other. For example, frontend presentation changes may not result in extensive backend changes because GraphQL allows the data specification to be modified readily. You can simply define or modify the shape of the data to fit the needs of the application as it grows. This results in less potential development work.

These are only some of the benefits of GraphQL. In the next few sections, you'll learn how GraphQL is structured and the properties that make it a unique alternative to REST.

Components of a GraphQL API

A standard GraphQL API is composed of a single schema that handles the shape of the data that will be queried. Your schema is linked to one or more of your data sources like a database or Lambda function. In between the two sits one or more resolvers that handle the business logic for your requests. Each component plays an important role in your GraphQL implementation. The following sections will introduce these three components and the role they play in the GraphQL service.



Topics

GraphQL schemas

The GraphQL schema is the foundation of a GraphQL API. It serves as the blueprint that defines the shape of your data. It's also a contract between your client and server that defines how your data will be retrieved and/or modified.

GraphQL schemas are written in the *Schema Definition Language* (SDL). SDL is composed of types and fields with an established structure:

- **Types**: Types are how GraphQL defines the shape and behavior of the data. GraphQL supports a multitude of types that will be explained later in this section. Each type that's defined in your schema will contain its own scope. Inside the scope will be one or more fields that can contain a value or logic that will be used in your GraphQL service. Types fill many different roles, the most common being objects or scalars (primitive value types).
- **Fields**: Fields exist within the scope of a type and hold the value that's requested from the GraphQL service. These are very similar to variables in other programming languages. The shape of the data you define in your fields will determine how the data is structured in a request/response operation. This allows developers to predict what will be returned without knowing how the backend of the service is implemented.

To visualize what a schema would look like, let's review the contents of a simple GraphQL schema. In production code, your schema will typically be in a file called `schema.graphql` or `schema.json`. Let's assume that we're peering into a project that implements a GraphQL service. This project is storing company personnel data, and the `schema.graphql` file is being used to retrieve personnel data and add new personnel to a database. The code may look like this:

schema.graphql

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

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

We can see that there are three types defined in the schema: `Person`, `Query`, and `Mutation`. Looking at `Person`, we can guess that this is the blueprint for an instance of a company employee, which would make this type an object. Inside its scope, we see `id`, `name`, and `age`. These are the fields that define the properties of a `Person`. This means our data source stores each `Person`'s `name` as a `String` scalar (primitive) type and `age` as an `Int` scalar (primitive) type. The `id` acts as a special, unique identifier for each `Person`. It's also a required value as denoted by the `!` symbol.

The next two object types behave differently. GraphQL reserves a few keywords for special object types that define how the data will be populated in the schema. A `Query` type will retrieve data from the source. In our example, our query might retrieve `Person` objects from a database. This may remind you of `GET` operations in RESTful terminology. A `Mutation` will modify data. In our example, our mutation may add more `Person` objects to the database. This may remind you of state-changing operations like PUT or POST. The behaviors of all special object types will be explained later in this section.

Let's assume the `Query` in our example will retrieve something from the database. If we look at the fields of `Query`, we see one field called `people`. Its field value is `[Person]`. This means we want to retrieve some instance of `Person` in the database. However, the addition of brackets means that we want to return a list of all `Person` instances and not just a specific one.

The `Mutation` type is responsible for performing state-changing operations like data modification. A mutation is responsible for performing some state-changing operation on the data source. In our example, our mutation contains an operation called `addPerson` that adds a new `Person` object to the database. The mutation uses a `Person` and expects an input for the `id`, `name`, and `age` fields.

At this point, you may be wondering how operations like `addPerson` work without a code implementation given that it supposedly performs some behavior and looks a lot like a function with a function name and parameters. Currently, it won't work because a schema only serves as the declaration. To implement the behavior of `addPerson`, we would have to add a resolver to it. A resolver is a unit of code that is executed whenever its associated field (in this case, the `addPerson`

operation) is called. If you want to use an operation, you'll have to add the resolver implementation at some point. In a way, you can think of the schema operation as the function declaration and the resolver as the definition. Resolvers will be explained in a different section.

This example shows only the simplest ways a schema can manipulate data. You build complex, robust, and scalable applications by leveraging the features of GraphQL and AWS AppSync. In the next section, we'll define all of the different types and field behaviors you can utilize in your schema.

GraphQL types

GraphQL supports many different types. As you saw in the previous section, types define the shape or behavior of your data. They are the fundamental building blocks of a GraphQL schema.

Types can be categorized into inputs and outputs. Inputs are types that are allowed to be passed in as the argument for the special object types (`Query`, `Mutation`, etc.), whereas output types are strictly used to store and return data. A list of types and their categorizations are listed below:

- **Objects**: An object contains fields describing an entity. For instance, an object could be something like a `book` with fields describing its characteristics like `authorName`, `publishingYear`, etc. They are strictly output types.
- **Scalars**: These are primitive types like int, string, etc. They are typically assigned to fields. Using the `authorName` field as an example, it could be assigned the `String` scalar to store a name like "John Smith". Scalars can be both input and output types.
- **Inputs**: Inputs allow you to pass a group of fields as an argument. They are structured very similarly to objects, but they can be passed in as arguments to special objects. Inputs allow you to define scalars, enums, and other inputs in its scope. Inputs can only be input types.
- **Special objects**: Special objects perform state-changing operations and do the bulk of the heavy lifting of the service. There are three special object types: query, mutation, and subscription. Queries typically fetch data; mutations manipulate data; subscriptions open and maintain a two-way connection between clients and servers for constant communication. Special objects are neither input nor output given their functionality.
- **Enums**: Enums are predefined lists of legal values. If you call an enum, its values can only be what's defined in its scope. For example, if you had an enum called `trafficLights` depicting a list of traffic signals, it could have values like `redLight` and `greenLight` but not `purpleLight`. A real traffic light will only have so many signals, so you could use the enum to define them and force them to be the only legal values when referencing `trafficLight`. Enums can be both input and output types.

- **Unions/interfaces**: Unions allow you to return one or more things in a request depending on the data that was requested by the client. For example, if you had a `Book` type with a `title` field and an `Author` type with a `name` field, you could create a union between both types. If your client wanted to query a database for the phrase "Julius Caesar", the union could return *Julius Caesar* (the play by William Shakespeare) from the `Book` `title` and *Julius Caesar* (the author of *Commentarii de Bello Gallico*) from the `Author` `name`. Unions can only be output types.

  Interfaces are sets of fields that objects must implement. This is a bit similar to interfaces in programming languages like Java where you must implement the fields defined in the interface. For example, let's say you made an interface called `Book` that contained a `title` field. Let's say you later created a type called `Novel` that implemented `Book`. Your `Novel` would have to include a `title` field. However, your `Novel` could also include other fields not in the interface like `pageCount` or `ISBN`. Interfaces can only be output types.

The following sections will explain how each type works in GraphQL.

Objects

GraphQL objects are the main type you will see in production code. In GraphQL, you can think of an object as a grouping of different fields (similar to variables in other languages), with each field being defined by a type (typically a scalar or another object) that can hold a value. Objects represent a unit of data that can be retrieved/manipulated from your service implementation.

Object types are declared using the `Type` keyword. Let's modify our schema example slightly:

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

type Occupation {
  title: String
}
```

The object types here are `Person` and `Occupation`. Each object has its own fields with its own types. One feature of GraphQL is the ability to set fields to other types. You can see the `occupation` field in `Person` contains an `Occupation` object type. We can make this association because GraphQL is only describing the data and not the implementation of the service.

Scalars

Scalars are essentially primitive types that hold values. In AWS AppSync, there are two types of scalars: the default GraphQL scalars and AWS AppSync scalars. Scalars are typically used to store field values within object types. Default GraphQL types include `Int`, `Float`, `String`, `Boolean`, and `ID`. Let's use the previous example again:

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

type Occupation {
  title: String
}
```

Singling out the `name` and `title` fields, both hold a `String` scalar. `Name` could return a string value like "`John  Smith`" and the title could return something like "`firefighter`". Some GraphQL implementations also support custom scalars using the `Scalar` keyword and implementing the type's behavior. However, AWS AppSync currently **doesn't support** custom scalars. For a list of scalars, see Scalar types in AWS AppSync.

Inputs

Due to the concept of input and output types, there are certain restrictions in place when passing in arguments. Types that commonly need to be passed in, especially objects, are restricted. You can use the input type to bypass this rule. Inputs are types that contain scalars, enums, and other input types.

Inputs are defined using the `input` keyword:

```
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
}
```

As you can see, we can have separate inputs that mimic the original type. These inputs will often be used in your field operations like this:

```
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
}
```

Note how we're still passing `occupationInput` in place of `Occupation` to create a `Person`.

This is but one scenario for inputs. They don't necessarily need to copy objects 1:1, and in production code, you most likely won't be using it like this. It's good practice to take advantage of GraphQL schemas by defining only what you need to input as arguments.

Also, the same inputs can be used in multiple operations, but we don't recommend doing this. Each operation should ideally contain its own unique copy of the inputs in case the schema's requirements change.

Special objects

GraphQL reserves a few keywords for special objects that define some of the business logic for how your schema will retrieve/manipulate data. At most, there can be one of each of these keywords in a schema. They act as entry points for all requested data that your clients run against your GraphQL service.

Special objects are also defined using the `type` keyword. Though they're used differently from regular object types, their implementation is very similar.

Queries

Queries are very similar to `GET` operations in that they perform a read-only fetch to get data from your source. In GraphQL, the `Query` defines all of the entry points for clients making requests against your server. There will always be a `Query` in your GraphQL implementation.

Here are the `Query` and modified object types we used in our previous schema example:

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

Our `Query` contains a field called `people` that returns a list of `Person` instances from the data source. Let's say we need to change the behavior of our application, and now we need to return a list of only the `Occupation` instances for some separate purpose. We could simply add it to the query:

```
type Query {
```

```
  people: [Person]
  occupations: [Occupation]
}
```

In GraphQL, we can treat our query as the single source of requests. As you can see, this is potentially much simpler than RESTful implementations that might use different endpoints to achieve the same thing (`.../api/1/people` and `.../api/1/occupations`).

Assuming we have a resolver implementation for this query, we can now perform an actual query. While the `Query` type exists, we have to explicitly call it for it to run in the application's code. This can be done using the `query` keyword:

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

As you can see, this query is called `getItems` and returns `people` (a list of `Person` objects) and `occupations` (a list of `Occupation` objects). In `people`, we're returning only the `name` field of each `Person`, while we're returning the `title` field of each `Occupation`. The response may look like this:

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

```
      },
      {
        "title": "Bookkeeper"
      },
      .
      .
      .
    ]
  }
}
```

The example response shows how the data follows the shape of the query. Each entry retrieved is listed within the scope of the field. `people` and `occupations` are returning things as separate lists. Though useful, it might be more convenient to modify the query to return a list of people's names and occupations:

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

This is a legal modification because our `Person` type contains an `occupation` field of type `Occupation`. When listed within the scope of `people`, we're returning each `Person`'s `name` along with their associated `Occupation` by `title`. The response may look like this:

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

```
      },
      .
      .
      .
    ]
  }
}
```

Mutations

Mutations are similar to state-changing operations like PUT or POST. They perform a write operation to modify data in the source, then fetch the response. They define your entry points for data modification requests. Unlike queries, a mutation may or may not be included in the schema depending on the project's needs. Here's the mutation from the schema example:

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

The `addPerson` field represents one entry point that adds a `Person` to the data source. `addPerson` is the field name; `id`, `name`, and `age` are the parameters; and `Person` is the return type. Looking back at the `Person` type:

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

We added the `occupation` field. However, we cannot set this field to `Occupation` directly because objects cannot be passed in as arguments; they are strictly output types. We should instead pass an input with the same fields as an argument:

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

We can also easily update our `addPerson` to include this as a parameter when making new `Person` instances:

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

Here's the updated schema:

```
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
}
```

Note that `occupation` will pass in the `title` field from `occupationInput` to complete the creation of the `Person` instead of the original `Occupation` object. Assuming we have a resolver implementation for `addPerson`, we can now perform an actual mutation. While the `Mutation` type exists, we have to explicitly call it for it to run in the application's code. This can be done using the `mutation` keyword:

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

This mutation is called `createPerson`, and `addPerson` is the operation. To create a new `Person`, we can enter the arguments for `id`, `name`, `age`, and `occupation`. In the scope of `addPerson`, we can also see other fields like `name`, `age`, etc. This is your response; these are the fields that will be returned after the `addPerson` operation is complete. Here's the final part of the example:

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

Using this mutation, a result might look like this:

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

As you can see, the response returned the values we requested in the same format that was defined in our mutation. It's good practice to return all values that were modified to reduce confusion and the need for more queries in the future. Mutations allow you to include multiple operations within its scope. They will be run sequentially in the order listed in the mutation. For example, if we create another operation called `add0ccupation` that adds job titles to the data source, we can call this in the mutation after `addPerson`. `addPerson` will be handled first followed by `add0ccupation`.

Subscriptions

Subscriptions use WebSockets to open a lasting, two-way connection between the server and its clients. Typically, a client will subscribe, or listen, to the server. Whenever the server makes a server-side change or performs an event, the subscribed client will receive the updates. This type of protocol is useful when multiple clients are subscribed and need to be notified about changes happening in the server or other clients. For instance, subscriptions can be used to update social media feeds. There could be two users, User A and User B, who are both subscribed to automatic notification updates whenever they receive direct messages. User A on Client A could send a direct message to User B on Client B. User A's client would send the direct message, which would be processed by the server. The server would then send the direct message to User B's account while sending an automatic notification to Client B.

Here's an example of a `Subscription` that we could add to the schema example:

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

The `personAdded` field will send a message to subscribed clients whenever a new `Person` is added to the data source. Assuming we have a resolver implementation for `personAdded`, we can now use the subscription. While the `Subscription` type exists, we have to explicitly call it for it to run in the application's code. This can be done using the `subscription` keyword:

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

The subscription is called `personAddedOperation`, and the operation is `personAdded`. `personAdded` will return the `id` and `name` fields of new `Person` instances. Looking at the mutation example, we added a `Person` using this operation:

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

If our clients were subscribed to updates to the newly added `Person`, they might see this after `addPerson` runs:

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

Below is a summary of what subscriptions offer:

Subscriptions are two-way channels that allow the client and server to receive quick, but steady, updates. They typically use the WebSocket protocol, which creates standardized and secure connections.

Subscriptions are nimble in that they reduce connection setup overhead. Once subscribed, a client can just keep running on that subscription for long periods of time. They generally use computing resources efficiently by allowing developers to tailor the lifetime of the subscription and to configure what information will be requested.

In general, subscriptions allow the client to make multiple subscriptions at once. As it pertains to AWS AppSync, subscriptions are only used for receiving real-time updates from the AWS AppSync service. They cannot be used to perform queries or mutations.

The main alternative to subscriptions is polling, which sends queries at set intervals to request data. This process is typically less efficient than subscriptions and puts a lot of strain on both the client and the backend.

One thing that wasn't mentioned in our schema example was the fact that your special object types must also be defined in a `schema` root. So when you export a schema in AWS AppSync, it might look like this:

schema.graphql

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

```
.
.
.

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

Enumerations

Enumerations, or enums, are special scalars that limit the legal arguments a type or field may have. This means that whenever an enum is defined in the schema, its associated type or field will be limited to the values in the enum. Enums are serialized as string scalars. Note that different programming languages may handle GraphQL enums differently. For example, JavaScript has no native enum support, so the enum values may be mapped to int values instead.

Enums are defined using the `enum` keyword. Here's an example:

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

When calling the `trafficLights` enum, the argument(s) can only be `solidRed`, `solidYellow`, `solidGreen`, etc. It's common to use enums to depict things that have a distinct but limited number of choices.

Unions/Interfaces

See Interfaces and unions in GraphQL.

GraphQL fields

Fields exist within the scope of a type and hold the value that's requested from the GraphQL service. These are very similar to variables in other programming languages. For example, here's a `Person` object type:

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

The fields in this case are `name` and `age` and hold a `String` and `Int` value, respectively. Object fields like the ones shown above can be used as the inputs in the fields (operations) of your queries and mutations. For example, see the `Query` below:

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

The `people` field is requesting all instances of `Person` from the data source. When you add or retrieve a `Person` in your GraphQL server, you can expect the data to follow the format of your types and fields, that is, the structure of your data in the schema determines how it'll be structured in your response:

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

```
}
```

Fields play an important role in structuring data. There are a couple of additional properties explained below that can be applied to fields for more customization.

Lists

Lists return all items of a specified type. A list can be added to a field's type using brackets `[]`:

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

In `Query`, the brackets surrounding `Person` indicate that you want to return all instances of `Person` from the data source as an array. In the response, the `name` and `age` values of each `Person` will be returned as a single, delimited list:

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

You aren't limited to special object types. You can also use lists in the fields of regular object types.

Non-nulls

Non-nulls indicate a field that cannot be null in the response. You can set a field to non-null by using the ! symbol:

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

The name field cannot be explicitly null. If you were to query the data source and provided a null input for this field, an error would be thrown.

You can combine lists and non-nulls. Compare these queries:

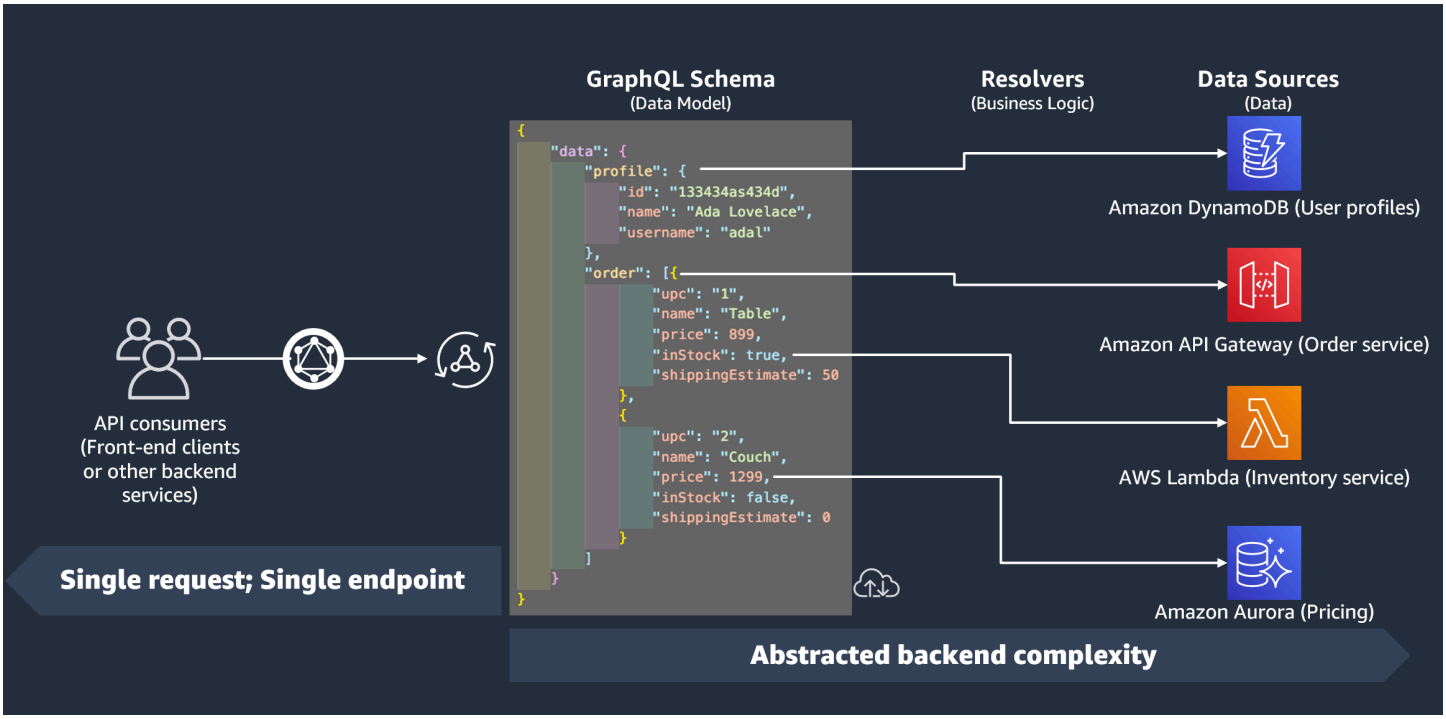
```
type Query {
  people: [Person!]      # Use case 1
}

.
.
.

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

.
.
.

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

In use case 1, the list cannot contain null items. In use case 2, the list itself cannot be set to null. In use case 3, the list and its items cannot be null. However, in any case, you can still return empty lists.

As you can see, there are many moving components in GraphQL. In this section, we showed the structure of a simple schema and the different types and fields a schema supports. In the following section, you will discover the other components of a GraphQL API and how they work with the schema.

Data sources

In the previous section, we learned that a schema defines the shape of your data. However, we never explained where that data came from. In real projects, your schema is like a gateway that handles all requests made to the server. When a request is made, the schema acts as the single endpoint that interfaces with the client. The schema will access, process, and relay data from the data source back to the client. See the infographic below:



AWS AppSync and GraphQL superbly implement Backend For Frontend (BFF) solutions. They work in tandem to reduce complexity at scale by abstracting the backend. If your service uses different data sources and/or microservices, you can essentially abstract some of the complexity away by defining the shape of the data of each source (subgraph) in a single schema (supergraph). This means your GraphQL API is not limited to using one data source. You can associate any number of data sources with your GraphQL API and specify in your code how they will interact with the service.

As you can see in the infographic, the GraphQL schema contains all of the information clients need to request data. This means everything can be processed in a single request rather than

multiple requests as is the case with REST. These requests go through the schema, which is the sole endpoint of the service. When requests are processed, a resolver (explained in the next section) executes its code to process the data from the relevant data source. When the response is returned, the subgraph tied to the data source will be populated with the data in the schema.

AWS AppSync supports many different data source types. In the table below, we'll describe each type, list some of the benefits of each, and provide useful links for additional context.

Data source	Description	Benefits	Supplemental information

Amazon DynamoDB	"Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. DynamoDB lets you offload the administrative burdens of operating and scaling a distributed database so that you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling. DynamoDB also offers encryption at rest, which eliminates the operational burden and complexity	• **Performance at scale**: DynamoDB is designed around consistent performance at any scale. This is possible through the use of partitions. DynamoDB will automatically partition your tables into several allocations that will be stored in multiple SSDs across several nodes. This will generally increase network throughput and reduce latency.
		• **Capacity at scale**: DynamoDB monitors your traffic and allows you to automatic
			• DynamoDB official documentation
			  • Partitions
			  • Auto scaling
			  • Fault tolerance
			  • Monitoring
			  • Security
			• GraphQL and DynamoDB
			• Resolver operations for DynamoDB
			• Pricing model

Data source	Description	Benefits	Supplemental information
	involved in protecting sensitive data."	ally scale your throughput if the network remains overloaded for extended periods of time. • **Availability and fault tolerance** : DynamoDB is supported by several physically isolated Regions, each containing several physically isolated Availability Zones. DynamoDB will automatic ally switch to a backup zone in the event of a service disruption. You can also back up and replicate your data manually for data assurance. • **Logging and monitoring**: DynamoDB provides several analytical tools for your tables. You can monitor your table's performan	

Data source	Description	Benefits	Supplemental information
		ce and create alarms to notify you of drastic changes to the service. • **Security**: DynamoDB follows strict protocols to ensure your data complies with your organization's security requirements. • **Integration with AWS AppSync**: DynamoDB is seamlessly integrated with our service. You can create new DynamoDB tables and automatically generate a schema from them to streamline your development process. We also provide an entire collection of operations to easily request data from existing DynamoDB tables	

Data source	Description	Benefits	Supplemental information
		in your account in your resolver.	

Data source	Description	Benefits	Supplemental information

AWS Lambda	"AWS Lambda is a compute service that lets you run code without provisioning or managing servers.
	
	Lambda runs your code on a high-avai lability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenan ce, capacity provision ing and automatic scaling, and logging. With Lambda, all you need to do is supply your code in one of the language runtimes that Lambda supports."
		• **Pay-as-you-use model**: Lambda only charges you when you use their resources. They also allow you to scale the amount of resources used with your applicati on needs.
		• **Automatic scaling**: Sometimes your application may require extra computing power for a particular process. Lambda allows you to automatically scale computing resources to fit the needs of your application.
		• **Faster deploymen t times**: You can streamline your developme nt process via a deployment package. Use a package to upload your function code to the Lambda
			• Official documenta tion
			  • Scaling
			  • deployment
			  • runtimes
			• Lambda resolver tutorial
			• Pricing model

Data source	Description	Benefits	Supplemental information
		service. You can then use their runtime environments to test and execute your functions. • **Versatility**: Lambda can be used in a multitude of use cases. You can seamlessly integrate Lambda with third-party services and AWS services alike. Some examples include CI/CD pipelines and mass mailing services. • **Integration with AWS AppSync**: You can easily invoke your Lambda functions in your resolver to handle requests. Our service provides a streamlined request operation to perform Lambda calls. We allow both single and batched calls.	

Data source	Description	Benefits	Supplemental information
OpenSearch	"Amazon OpenSearch Service is a managed service that makes it easy to deploy, operate, and scale OpenSearch clusters in the AWS Cloud. Amazon OpenSearch Service supports OpenSearch and legacy Elasticsearch OSS (up to 7.10, the final open-source version of the software). When you create a cluster, you have the option of which search engine to use. **OpenSearch** is a fully open-source search and analytics engine for use cases such as log analytics, real-time application monitoring, and clickstream analysis. For more information, see the OpenSearch documentation.	• **Scaling**: You can easily scale the service to fit your service requirements through OpenSearch Serverless. • **Data ingestion**: You can use OpenSearch Ingestion to import, process, and analyze data. There are many applications for data ingestion, which you can find here. • **Security**: OpenSearch can manage your AWS security configuration including IAM, CloudTrail, VPCs, authentication, etc. • **Availability**: OpenSearch also supports different Regions and Availability Zones in its service. • **Integration with AWS AppSync**: In	• Official documentation • Serverless • Pricing model

Data source	Description	Benefits	Supplemental information
	Amazon OpenSearch Service provisions all the resources for your OpenSearch cluster and launches it. It also automatically detects and replaces failed OpenSearch Service nodes, reducing the overhead associated with self-managed infrastructures. You can scale your cluster with a single API call or a few clicks in the console."	AWS AppSync, you can use GraphQL APIs to store and retrieve data from existing OpenSearch Service domains in your account.	

Data source	Description	Benefits	Supplemental information
HTTP endpoints	You can use HTTP endpoints as data sources. AWS AppSync can send requests to the endpoints with the relevant informati on like params and payload. The HTTP response will be exposed to the resolver, which will return the final response after it finishes its operation (s).	• Useful for simple applications that aren't as integrate d with services like Lambda.	• Resolver reference

Data source	Description	Benefits	Supplemental information

Amazon EventBridge	"EventBridge is a serverless service that uses events to connect application components together, making it easier for you to build scalable event-driven applications. Use it to route events from sources such as home-grown applications, AWS services, and third-party software to consumer applications across your organization. EventBridge provides a simple and consistent way to ingest, filter, transform, and deliver events so you can build new applications quickly."	• **Event-driven architecture**: You can take advantage of event-driven architecture.
		• **Scheduling**: You can use the EventBridge Scheduler to automate your tasks and rules using cron expressions or set time intervals as an alternative to event patterns.
		• **Pipes**: Using EventBridge Pipes, you can replace the event bus with a pipe that includes additional filtering event patterns and enrichment through data transforms before sending the event to the target.
		• **Integration with AWS AppSync**: AWS AppSync allows you to send events to event
			• Official documentation
			  • Pipes
			  • Scheduler
			• Resolver reference
			• Pricing model

Data source	Description	Benefits	Supplemental information
		buses using your resolver.	

Data source	Description	Benefits	Supplemental information

Relational databases	"Amazon Relational Database Service (Amazon RDS) is a web service that makes it easier to set up, operate, and scale a relational database in the AWS Cloud. It provides cost-efficient, resizable capacity for an industry-standard relational database and manages common database administration tasks."	• **Managing made easy**: Periodically, RDS performs maintenance on its resources. Maintenance most often involves updates to the DB instance's underlying hardware, underlying operating system (OS), or database engine version. Under normal circumstances, you can decide when to perform updates (exceptions include security patches).
		• **Recommendations**: RDS' recommendation feature provides automated suggestions for fixing potential issues in your instance.
		• **Availability**: RDS is available in different physical Regions across the
			• Official documentation
			  • Features
			  • Maintenance
			  • Recommendations
			  • Storage options
			  • Availability
			  • Security
			• Pricing model

Data source	Description	Benefits	Supplemental information
		world. You can easily distribut e your database needs across different nodes to provide better service to your customers. • **Customisation**: RDS is tailored to meet the requirements of large corporati ons. RDS provides various options for computing, quick deploymen t, scalability, and storage. • **Security**: RDS is integrated with several tools and services to maintain database security on the user, database, and network levels. • **Integration with AWS AppSync**: If you're looking for a mature backend solution, AWS AppSync	

Data source	Description	Benefits	Supplemental information
		allows you to send, process, store, and return data using your instance as the data source.	
None data source	If you aren't planning on using a data source service, you can set it to none. A none data source, while still explicitly categorized as a data source, isn't a storage medium. Despite that, it's still useful in certain instances for data manipulation and pass-throughs.	• Potentially useful for things like data conversion • Useful when resolving something locally	• Resolver reference

Tip

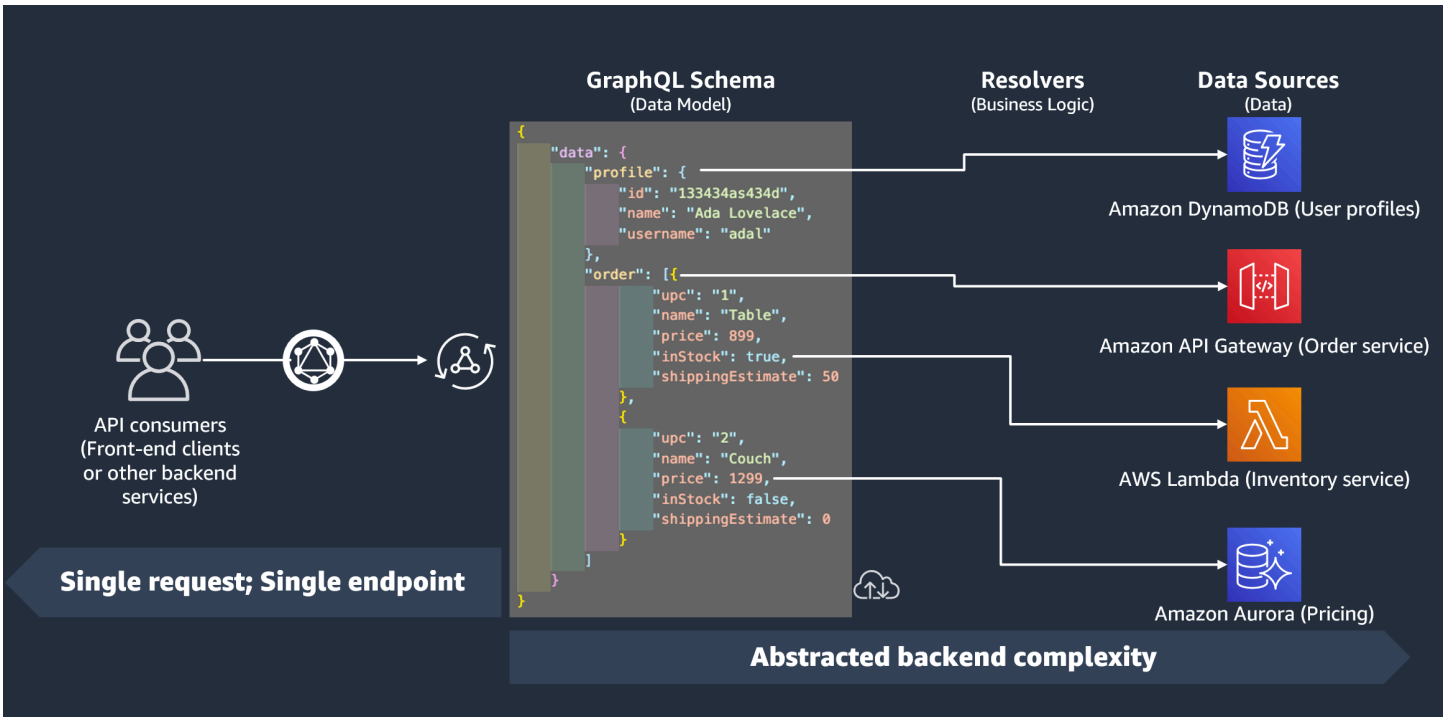
For more information about how data sources interact with AWS AppSync, see Attaching a data source.

Resolvers

From the previous sections, you learned about the components of the schema and data source. Now, we need to address how the schema and data sources interact. It all begins with the resolver.

A resolver is a unit of code that handles how that field's data will be resolved when a request is made to the service. Resolvers are attached to specific fields within your types in your schema. They are most commonly used to implement the state-changing operations for your query,

mutation, and subscription field operations. The resolver will process a client's request, then return the result, which can be a group of output types like objects or scalars:



Resolver runtime

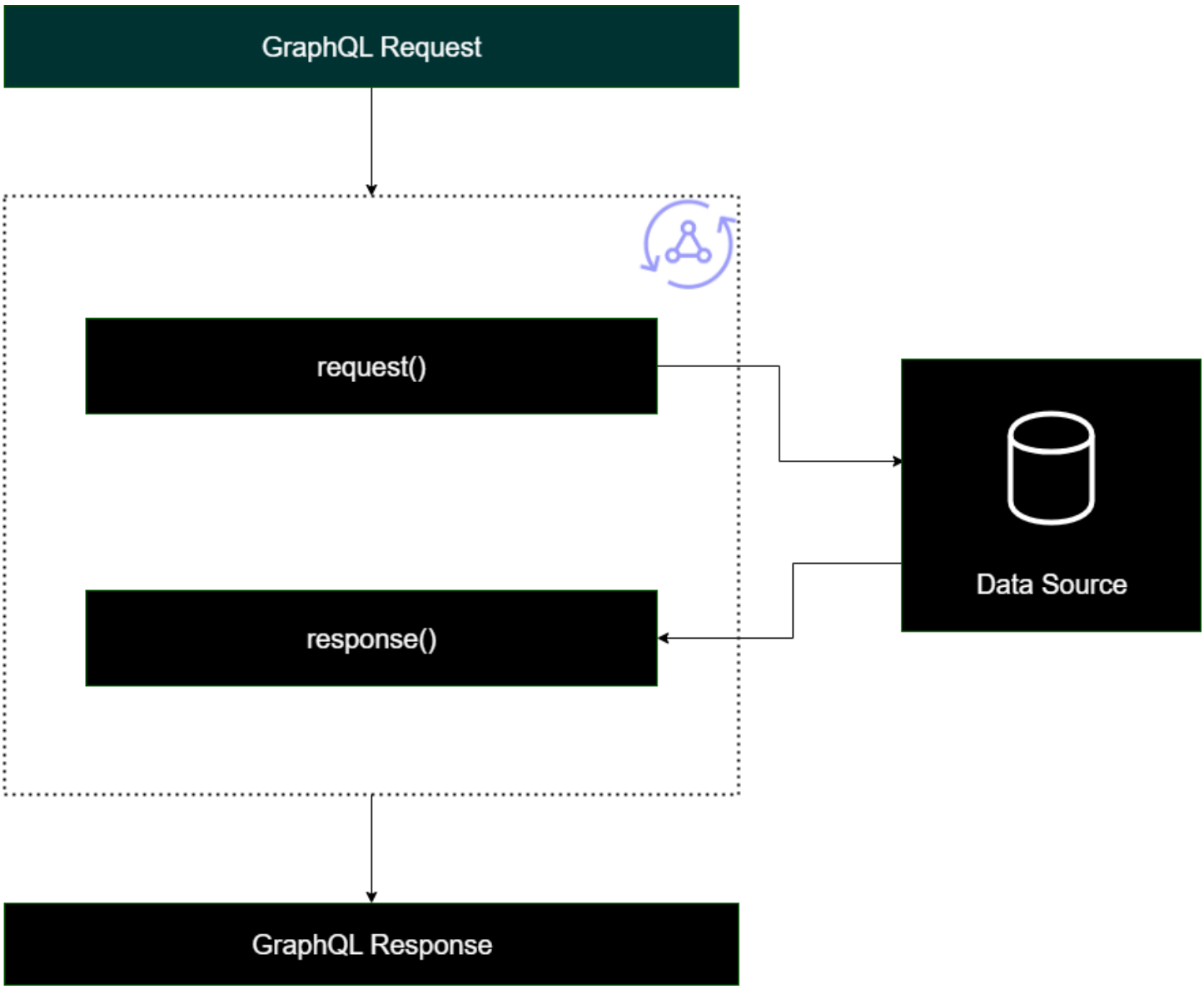
In AWS AppSync, you must first specify a runtime for your resolver. A resolver runtime indicates the environment in which a resolver is executed. It also dictates the language your resolvers will be written in. AWS AppSync currently supports APPSYNC_JS for JavaScript and Velocity Template Language (VTL). See JavaScript runtime features for resolvers and functions for JavaScript or Resolver mapping template utility reference for VTL.

Resolver structure

Code-wise, resolvers can be structured in a couple of ways. There are **unit** and **pipeline** resolvers.

Unit resolvers

A unit resolver is composed of code that defines a single request and response handler that are executed against a data source. The request handler takes a context object as an argument and returns the request payload used to call your data source. The response handler receives a payload back from the data source with the result of the executed request. The response handler transforms the payload into a GraphQL response to resolve the GraphQL field.



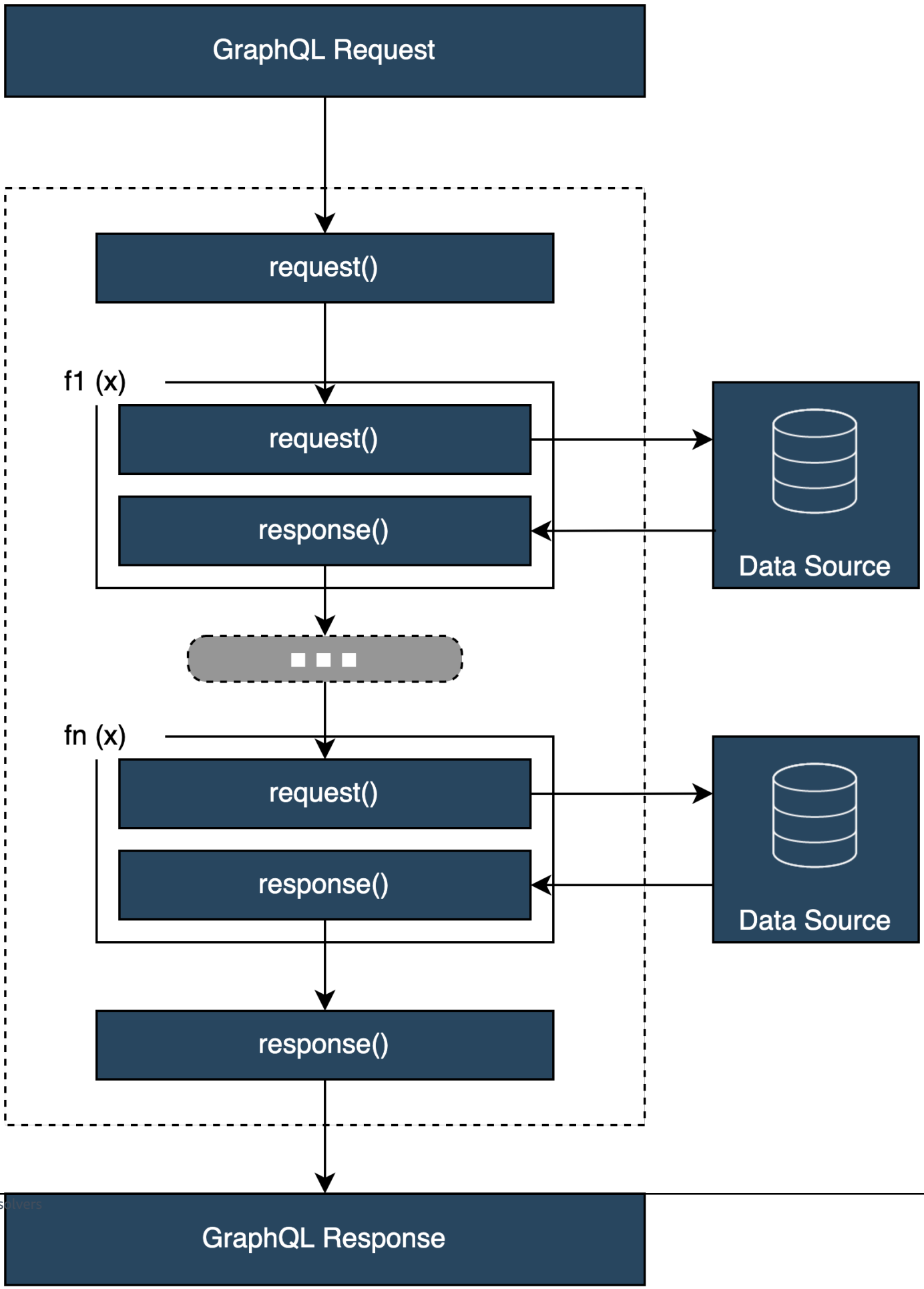
Pipeline resolvers

When implementing pipeline resolvers, there is a general structure they follow:

- **Before step**: When a request is made by the client, the resolvers for the schema fields being used (typically your queries, mutations, subscriptions) are passed the request data. The resolver will begin processing the request data with a before step handler, which allows some preprocessing operations to be performed before the data moves through the resolver.
- **Function(s)**: After the before step runs, the request is passed to the functions list. The first function in the list will execute against the data source. A function is a subset of your resolver's code containing its own request and response handler. A request hander will take the request

data and perform operations against the data source. The response handler will process the data source's response before passing it back to the list. If there is more than one function, the request data will be sent to the next function in the list to be executed. Functions in the list will be executed serially in the order defined by the developer. Once all functions have been executed, the final result is passed to the after step.

- **After step**: The after step is a handler function that allows you to perform some final operations on the final function's response before passing it to the GraphQL response.


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

Resolver handler structure

Handlers are typically functions called `Request` and `Response`:

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

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

In a unit resolver, there will only be one set of these functions. In a pipeline resolver, there will be a set of these for the before and after step and an additional set per function. To visualize how this could look, let's review a simple `Query` type:

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

This is a simple query with one field called `helloWorld` of type `String`. Let's assume we always want this field to return the string "Hello World". To implement this behavior, we need to add the resolver to this field. In a unit resolver, we could add something like this:

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

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

The `request` can just be left blank because we're not requesting or processing data. We can also assume our data source is None, indicating this code doesn't need to perform any invocations. The response simply returns "Hello World". To test this resolver, we need to make a request using the query type:

```
query helloWorldTest {
  helloWorld
}
```

This is a query called `helloWorldTest` that returns the `helloWorld` field. When executed, the `helloWorld` field resolver also executes and returns the response:

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

Returning constants like this is the simplest thing you could do. In reality, you'll be returning inputs, lists, and more. Here's a more complicated example:

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

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

Here we're returning a list of `Books`. Let's assume we're using a DynamoDB table to store book data. Our handlers may look like this:

```
/**
 * 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;
}
```

Our request used a built-in scan operation to search for all entries in the table, stored the findings in the context, then passed it to the response. The response took the result items and returned them in the response:

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

        ...

      ]
    }
  }
}
```

Resolver context

In a resolver, each step in the chain of handlers must be aware of the state of the data from the previous steps. The result from one handler can be stored and passed to another as an argument. GraphQL defines four basic resolver arguments:

Resolver base arguments	Description
`obj`, `root`, `parent`, etc.	The result of the parent.
`args`	The arguments provided to the field in the GraphQL query.
`context`	A value which is provided to every resolver and holds important contextual information like the currently logged in user, or access to a database.

Resolver base arguments	Description
`info`	A value which holds field-specific informati on relevant to the current query as well as the schema details.

In AWS AppSync, the `context` (ctx) argument can hold all of the data mentioned above. It's an object that's created per request and contains data like authorization credentials, result data, errors, request metadata, etc. The context is an easy way for programmers to manipulate data coming from other parts of the request. Take this snippet again:

```
/**
 * 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;
}
```

The request is given the context (ctx) as the argument; this is the state of the request. It performs a scan for all items in a table, then stores the result back in the context in `result`. The context is then passed to the response argument, which accesses the `result` and returns its contents.

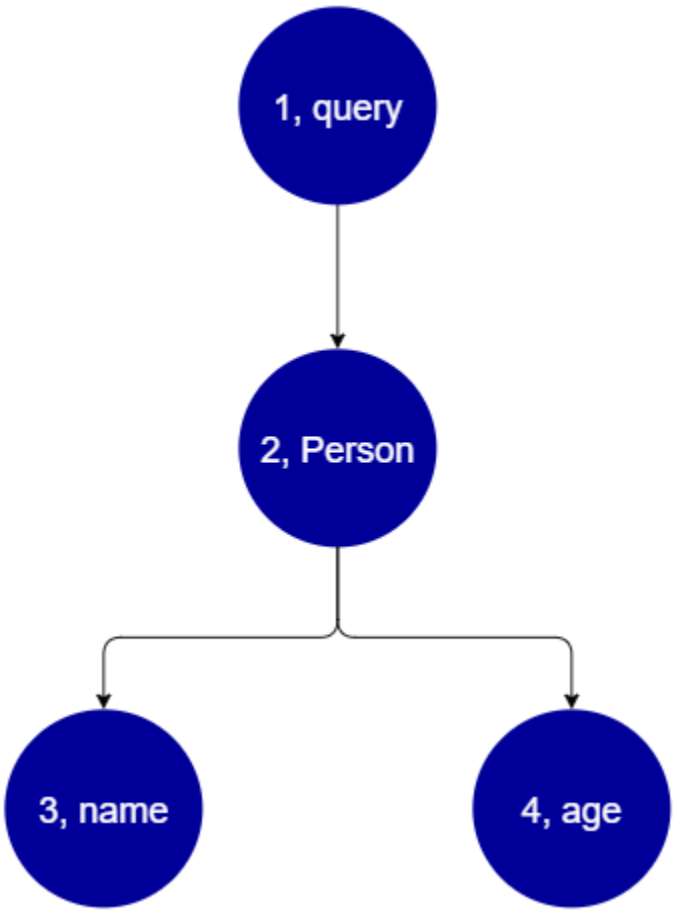
Requests and Parsing

When you make a query to your GraphQL service, it must run through a parsing and validation process before being executed. Your request will be parsed and translated into an abstract syntax tree. The content of the tree is validated by running through several validation algorithms against your schema. After the validation step, the nodes of the tree are traversed and processed. Resolvers are invoked, the results are stored in the context, and the response is returned. For example, take this query:

```
query {
```

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

We're returning `Person` with a `name` and `age` fields. When running this query, the tree will look something like this:



From the tree, it appears that this request will search the root for the `Query` in the schema. Inside of the query, the `Person` field will be resolved. From previous examples, we know that this could be an input from the user, a list of values, etc. `Person` is most likely tied to an object type holding the fields we need (`name` and `age`). Once these two child fields are found, they are resolved in the order given (`name` followed by `age`). Once the tree is completely resolved, the request is completed and will be sent back to the client.

Additional properties of GraphQL

GraphQL consists of several design principles to maintain simplicity and robustness at scale.

Declarative

GraphQL is declarative, which means the user will describe (shape) the data by only declaring the fields they want to query. The response will only return the data for these properties. For example, here's an operation that retrieves a Book object in a DynamoDB table with the ISBN 13 `id` value of *`9780199536061`*:

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

The response will return the fields in the payload (`name`, `year`, and `author`) and nothing else:

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

Because of this design principle, GraphQL eliminates the perennial issues of over- and under-fetching that REST APIs deal with in complex systems. This results in more efficient data gathering and improved network performance.

Hierarchical

GraphQL is flexible in that the data requested can be shaped by the user to fit the needs of the application. Requested data always follows the types and syntax of the properties defined in

your GraphQL API. For instance, the following snippet shows the `getBook` operation with a new field scope called `quotes` that returns all stored quote strings and pages linked to the Book *9780199536061*:

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

Running this query returns the following result:

```
{
  "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
         }
      ]
    }
  }
```

```
}
```

As you can see, the `quotes` fields linked to the requested book was returned as an array in the same format that was described by our query. Although it wasn't shown here, GraphQL has the added advantage of not being particular about the location of the data it's retrieving. `Books` and `quotes` could be stored separately, but GraphQL will still retrieve the information so long as the association exists. This means your query can retrieve multitudes of standalone data in a single request.

Introspective

GraphQL is self-documenting, or introspective. It supports several built-in operations that allow users to view the underlying types and fields within the schema. For example, here's a `Foo` type with a `date` and `description` field:

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

We could use the `_type` operation to find the typing metadata underneath the schema:

```
{
  __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
      }
    }
  }
}
```

This will return a response:

```
{
  "__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
      },
    ]
  }
}
```

This feature can be used to find out what types and fields a particular GraphQL schema supports. GraphQL supports a wide variety of these introspective operations. For more information, see Introspection.

Strong typing

GraphQL supports strong typing through its types and fields system. When you define something in your schema, it must have a type that can be validated before runtime. It must also follow GraphQL's syntax specification. This concept is no different from programming in other languages. For example, here's the Foo type from earlier:

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

We can see that Foo is the object that will be created. Inside an instance of Foo, there will be a `date` and `description` field, both of the `String` primitive type (scalar). Syntactically, we see that Foo was declared, and its fields exist inside its scope. This combination of type checking and logical syntax ensures that your GraphQL API is concise and self-evident. GraphQL's typing and syntax specification can be found here.

Getting started: Creating your first GraphQL API in AWS AppSync

You can use the AWS AppSync console to configure and launch a GraphQL API. GraphQL APIs generally require three components:

1. **GraphQL schema** - Your GraphQL schema is the blueprint of the API. It defines the types and fields that you can request when an operation is executed. To populate the schema with data, you must connect data sources to the GraphQL API. In this quickstart guide, we'll be creating a schema using a predefined model.
2. **Data sources**- These are the resources that contain the data for populating your GraphQL API. This can be a DynamoDB table, Lambda function, etc. AWS AppSync supports a multitude of data sources to build robust and scalable GraphQL APIs. Data sources are linked to fields in the schema. Whenever a request is performed on a field, the data from the source populates the field. This mechanism is controlled by the resolver. In this quickstart guide, we'll be creating a data source using a predefined model alongside the schema.
3. **Resolvers** - Resolvers are responsible for linking the schema field to the data source. They retrieve the data from the source, then return the result based on what was defined by the field. AWS AppSync supports both JavaScript and VTL for writing resolvers for your GraphQL APIs. In this quickstart guide, the resolvers will be automatically generated based on the schema and the data source. We won't be delving into this in this section.

AWS AppSync supports the creation and configuration of all GraphQL components. When you open the console, you can use the following methods to create your API:

1. Designing a customized GraphQL API by generating it through a predefined model and setting up a new DynamoDB table (data source) to support it.
2. Designing a GraphQL API with a blank schema and no data sources or resolvers.
3. Using a DynamoDB table to import data and generate your schema's types and fields.
4. Using AWS AppSync's WebSocket capabilities and Pub/Sub architecture to develop real-time APIs.
5. Using existing GraphQL APIs (source APIs) to link to a Merged API.

> **Note**
>
> We recommend reviewing the Designing a schema section before working with more advanced tools. These guides will explain simpler examples that you can use conceptually to build more complex applications in AWS AppSync.

AWS AppSync also supports several non-console options to create GraphQL APIs. These include:

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

The following example will show you how to create the basic components of a GraphQL API using predefined models and DynamoDB.

Topics

Launching a schema in the AWS AppSync console

In this example, you will create a Todo API that allows users to create Todo items for daily chore reminders like `Finish task` or `Pick up groceries`. This API will demonstrate how to use GraphQL operations where the state persists in a DynamoDB table.

Conceptually, there are three major steps to creating your first GraphQL API. You must define the schema (types and fields), attach your data source(s) to your field(s), then write the resolver that handles the business logic. However, the console experience changes the order of this. We will begin by defining how we want our data source to interact with our schema, then define the schema and resolver later.

To create your GraphQL API

1. Sign in to the AWS Management Console and open the AppSync console.
2. In the **Dashboard**, choose **Create API**.
3. While **GraphQL APIs** is selected, choose **Design from scratch**. Then, choose **Next**.
4. For **API name**, change the prepopulated name to **`Todo API`**, then choose **Next**.

   > **Note**
   >
   > There are also other options present here, but we won't be using them for this example.

5. In the **Specify GraphQL resources** section, do the following:

   a. Choose **Create type backed by a DynamoDB table now**.

   > **Note**
   >
   > This means we are going to create a new DynamoDB table to attach as a data source.

   b. In the **Model Name** field, enter **Todo**.

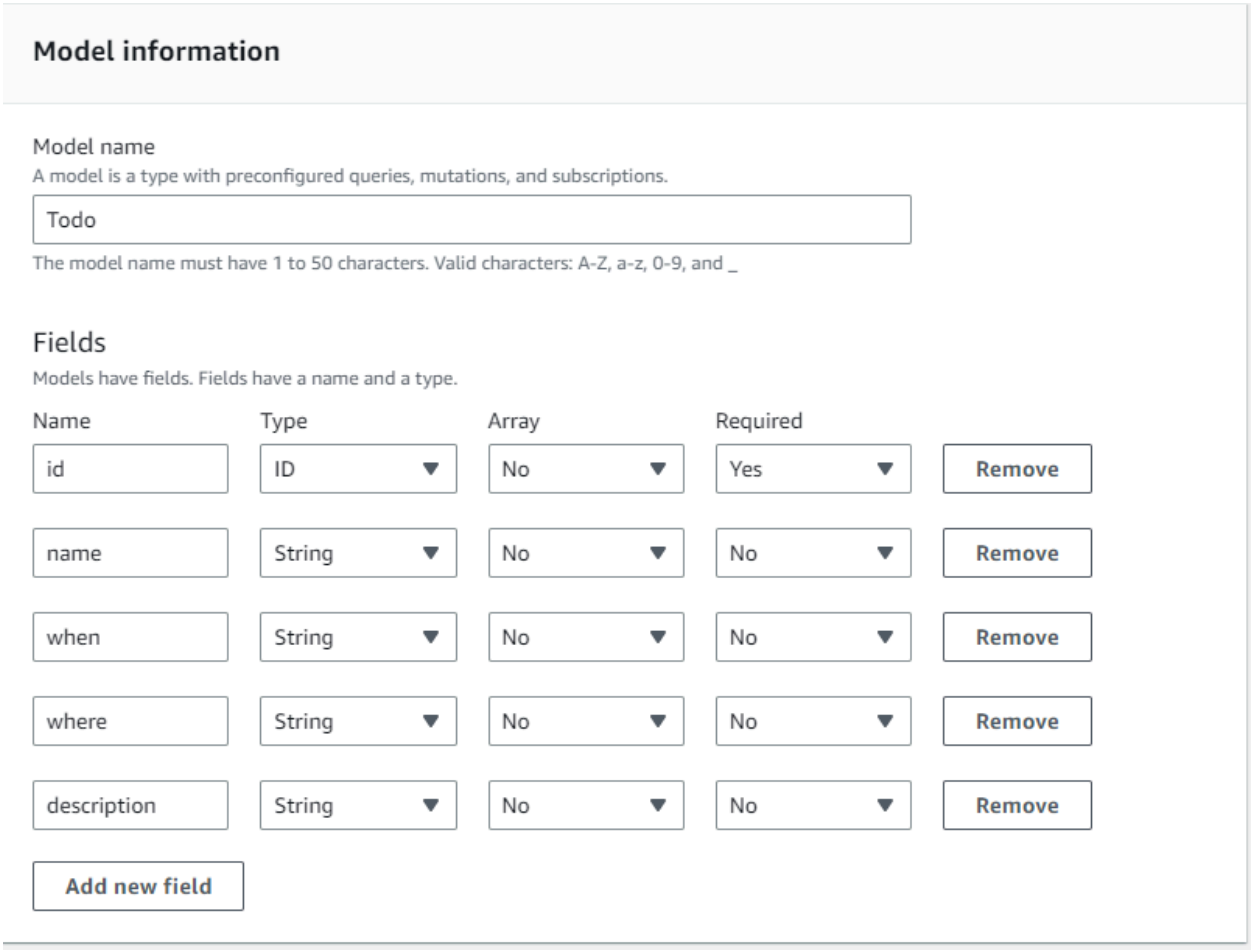
   > **Note**
   >
   > Our first requirement is to define our schema. This **Model Name** will be the type name, so what you're really doing is creating a `type` called `Todo` that will exist in the schema:
   >
   > ```
   > type Todo {}
   > ```

   c. Under **Fields**, do the following:

      i. Create a field named **`id`**, with the type `ID`, and required set to `Yes`.

> **Note**
>
> These are the fields that will exist within the scope of your `Todo` type. Your field name here will be called `id` with a type of `ID!`:
>
> ```
> type Todo {
>  id: ID!
> }
> ```
>
> AWS AppSync supports multiple scalar values for different use cases.

ii. Using **Add new field**, create four additional fields with the `Name` values set to **`name`**, **`when`**, **`where`**, and **`description`**. Their `Type` values will be `String`, and the `Array` and `Required` values will both be set to `No`. It will look like this:



> **Note**
>
> The full type and its fields will look like this:
>
> ```
> type Todo {
>  id: ID!
>  name: String
>  when: String
>  where: String
>  description: String
> }
> ```
>
> Because we're creating a schema using this predefined model, it will also be populated with several boilerplate mutations based on the type such as `create`, `delete`, and `update` to help you populate your data source easily.

d. Under **configure model table**, enter a table name, such as **`TodoAPITable`**. Set the **Primary Key** to `id`.

> **Note**
>
> We're essentially creating a new DynamoDB table called *`TodoAPITable`* that will be attached to the API as our primary data source. Our primary key is set to the required `id` field that we defined before this. Note that this new table is blank and doesn't contain anything except for the partition key.

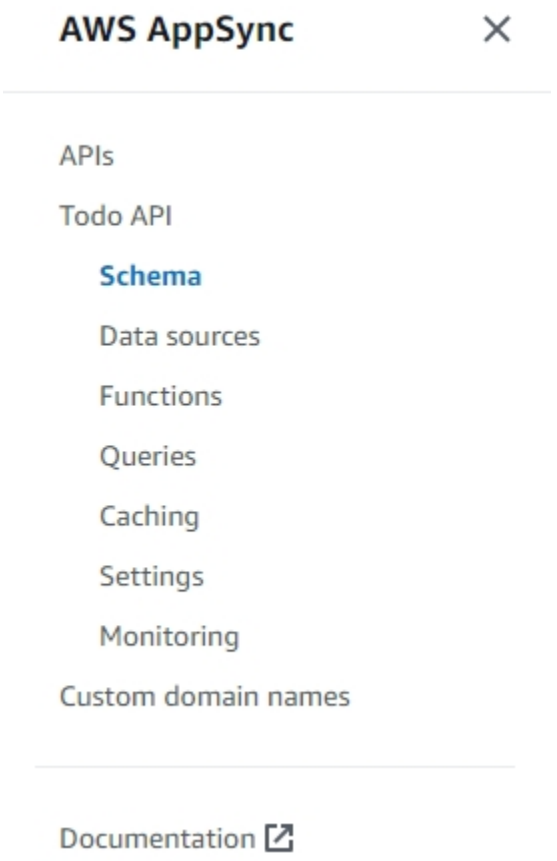
e. Choose **Next**.

6. Review your changes and choose **Create API**. Wait a moment to let the AWS AppSync service finish creating your API.

You have successfully created a GraphQL API with its schema and DynamoDB data source. To summarize the steps above, we chose to create a completely new GraphQL API. We defined the name of the API, then added our schema definition by adding our first type. We defined the type and its fields, then chose to attach a data source to one of the fields by creating a new DynamoDB table with no data in it.

Taking a tour of the AWS AppSync console

Before we add data to our DynamoDB table, we should review the basic features of the AWS AppSync console experience. The AWS AppSync console tab on the left-hand side of the page allows users to easily navigate to any of the major components or configuration options that AWS AppSync provides:



Schema designer

Choose **Schema** to view the schema you just created. If you review the schema's contents, you'll notice that it has already been loaded with a bunch of helper operations to streamline the development process. In the **Schema** editor, if you scroll through the code, you'll eventually reach the model you defined in the previous section:

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

```
  description: String
}
```

Your model became the base type that was used throughout your schema. We'll start adding data to our data source using mutations that were automatically generated from this type.

Here are some additional tips and facts about the **Schema** editor:

1. The code editor has linting and error-checking capabilities that you can use when writing your own apps.
2. The right side of the console shows the GraphQL types that have been created and resolvers on different top-level types, such as queries.
3. When adding new types to a schema (for example, `type User {...}`), you can have AWS AppSync provision DynamoDB resources for you. These include the proper primary key, sort key, and index design to best match your GraphQL data access pattern. If you choose **Create Resources** at the top and choose one of these user-defined types from the menu, you can choose different field options in the schema design. We will cover this in the design a schema section.

Resolver configuration

In the schema designer, the **Resolvers** section contains all of the types and fields in your schema. If you scroll through the list of fields, you'll notice that you can attach resolvers to certain fields by choosing **Attach**. This will open up a code editor in which you can write your resolver code. AWS AppSync supports both VTL and JavaScript runtimes, which can be changed at the top of the page by choosing **Actions**, then **Update Runtime**. At the bottom of the page, you can also create functions that will run several operations in a sequence. However, resolvers are an advanced topic, and we won't be covering that in this section.

Data sources

Choose **Data sources** to view your DynamoDB table. By choosing the `Resource` option (if available), you can view your data source's configuration. In our example, this leads to the DynamoDB console. From there, you can edit your data. You can also directly edit some of the data by choosing the data source, then choosing **Edit**. If you ever need to delete your data source, you can choose your data source, then select **Delete**. Lastly, you can create new data sources by choosing **Create data source**, then configuring the name and type. Note that this option is for

linking the AWS AppSync service to an existing resource. You still need to create the resource in your account using the relevant service before AWS AppSync recognizes it.

Queries

Choose **Queries** to view your queries and mutations. When we created our GraphQL API using our model, AWS AppSync automatically generated some helper mutations and queries for testing purposes. In the query editor, the left-hand side contains the **Explorer**. This is a list showing all of your mutations and queries. You can easily enable the operations and fields you want to use here by clicking on their name values. This will cause the code to appear automatically in the center part of the editor. Here, you can edit your mutations and queries by modifying values. At the bottom of the editor, you have the **Query Variable** editor that allows you to enter the field values for the input variables of your operations. Choosing **Run** at the top of the editor will bring up a drop-down list to select the query/mutation to run. The output for this run will appear on the right-hand side of the page. Back in the **Explorer** section at the top, you can choose an operation (Query, Mutation, Subscription), then choose the **+** symbol to add a new instance of that particular operation. At the top of the page, there will be another drop-down list that contains the authorization mode for your query runs. However, we will not be covering that feature in this section (For more information, see Security.).

Settings

Choose **Settings** to view some configuration options for your GraphQL API. Here, you can enable some options like logging, tracing, and web application firewall functionality. You can also add new authorization modes to protect your data from unwanted leaks to the public. However, these options are more advanced and will not be covered in this section.

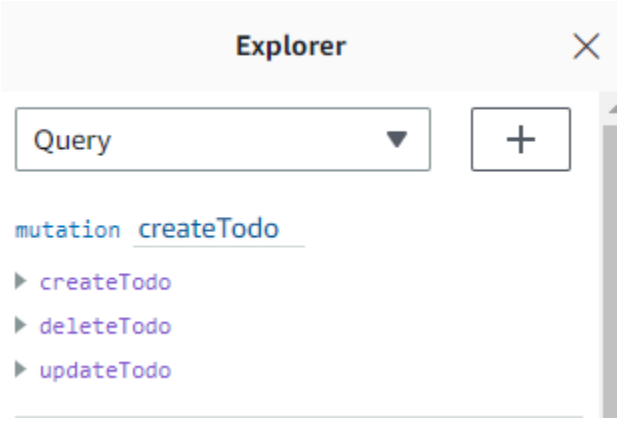
> **Note**
>
> The default authorization mode, `API_KEY`, uses an API key to test the application. This is the base authorization that's given to all newly created GraphQL APIs. We recommend that you use a different method for production. For the sake of the example in this section, we will only use the API key. For more information about the supported authorization methods, see Security.

Using GraphQL mutations to add data to a DynamoDB table in the AWS AppSync console

Your next step is to add data to your blank DynamoDB table using a GraphQL mutation. Mutations are one of the fundamental operation types in GraphQL. They are defined in the schema and allow you to manipulate data in your data source. In terms of REST APIs, these are very similar to operations like PUT or POST.

To add data to your data source

1. If you haven't already done so, sign in to the AWS Management Console and open the AppSync console.
2. Choose your API from the table.
3. In the tab to the left, choose **Queries**.
4. In the **Explorer** tab to the left of the table, you might see several mutations and queries already defined in the query editor:



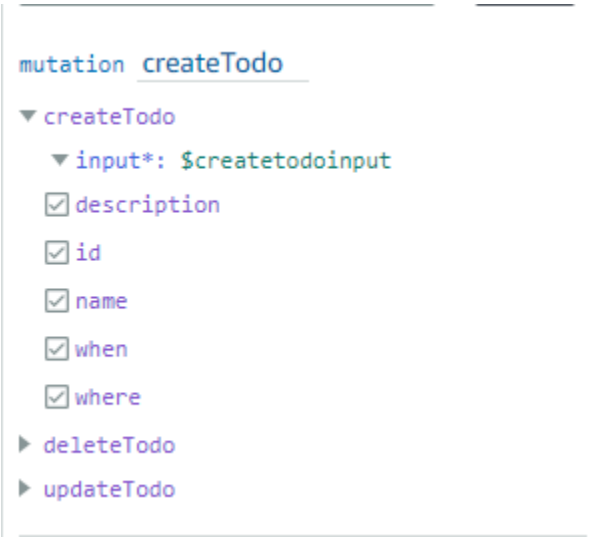
> **Note**
>
> This mutation is actually sitting in your schema as the `Mutation` type. It has the code:

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

As you can see, the operations here are similar to what's inside the query editor.

AWS AppSync automatically generated these from the model we defined earlier. This example will use the `createTodo` mutation to add entries to our *`TodoAPITable`* table.

5. Choose the `createTodo` operation by expanding it under the `createTodo` mutation:



Enable the checkboxes for all of the fields like the picture above.

> **Note**
>
> The attributes you see here are the different modifiable elements of the mutation. Your `input` can be thought of as the parameter of `createTodo`. The various options with checkboxes are the fields that will be returned in the response once an operation is performed.

6. In the code editor in the center of the screen, you'll notice that the operation appears underneath the `createTodo` mutation:

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

```
  }
}
```

Note

To explain this snippet properly, we must also look at the schema code. The declaration `mutation createTodo($createtodoinput: CreateTodoInput!){}` is the mutation with one of its operations, `createTodo`. The full mutation is located in the schema:

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

Going back to the mutation declaration from the editor, the parameter is an object called `$createtodoinput` with a required input type of `CreateTodoInput`. Note that `CreateTodoInput` (and all inputs in the mutation) are also defined in the schema. For example, here's the boilerplate code for `CreateTodoInput`:

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

It contains the fields we defined in our model, namely `name`, `when`, `where`, and `description`.
Going back to the editor code, in `createTodo(input: $createtodoinput) {}`, we declare the input as `$createtodoinput`, which was also used in the mutation declaration. We do this because this allows GraphQL to validate our inputs against the provided types and ensure that they are being used with the correct inputs.
The final part of the editor code shows the fields that will be returned in the response after an operation is performed:

```
{
    where
```

```
    when
    name
    id
    description
  }
```

In the **Query variables** tab below this editor, there will be a generic `createtodoinput` object that may have the following data:

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

Note

This is where we allocate the values for the input mentioned earlier:

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

Change the `createtodoinput` by adding information we want to put in our DynamoDB table. In this case, we wanted to create some Todo items as reminders:

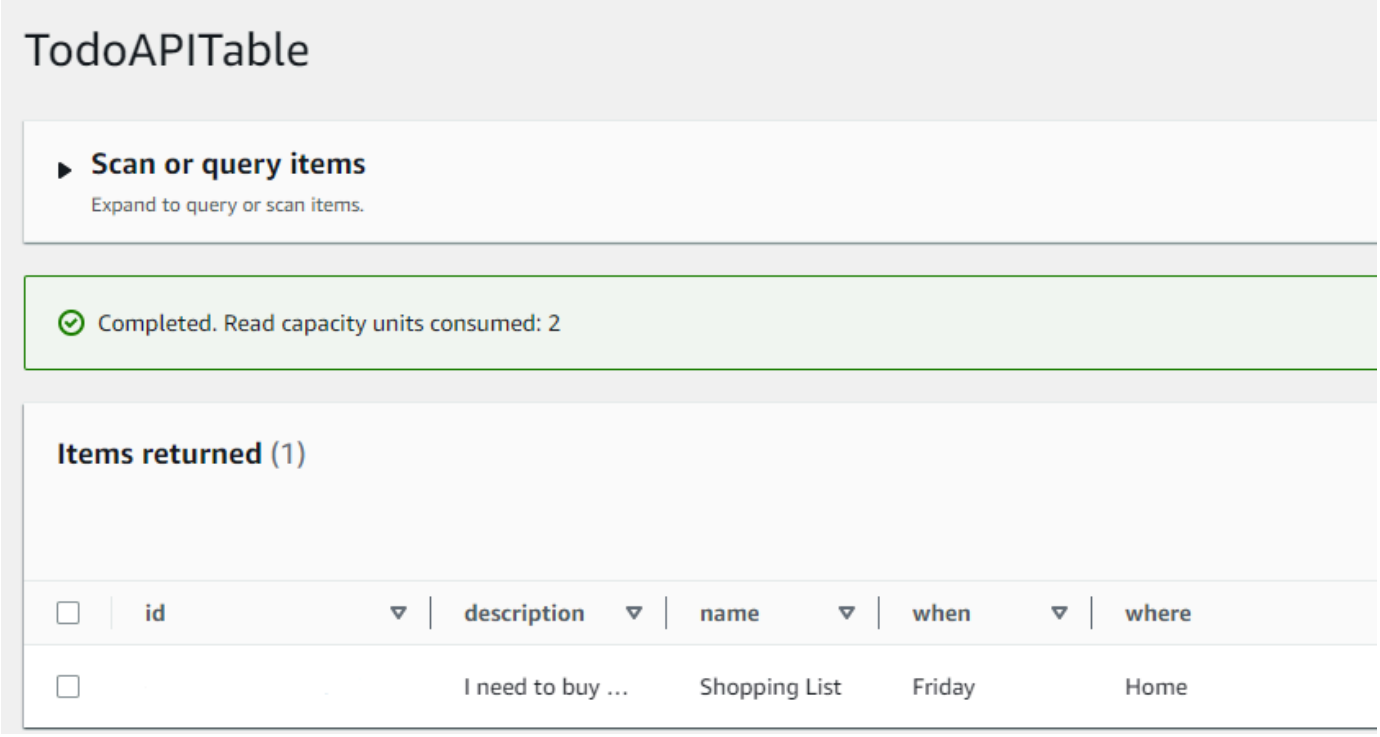
```
{
  "createtodoinput": {
    "name": "Shopping List",
    "when": "Friday",
    "where": "Home",
```

```
    "description": "I need to buy eggs"
  }
}
```

7. Choose **Run** at the top of the editor. Choose **createTodo** in the drop-down list. On the right-hand side of the editor, you should see the response. It may look something like this:

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

If you navigate to the DynamoDB service, you'll now see an entry in your data source with this information:



To summarize the operation, the GraphQL engine parsed the record, and a resolver inserted it into your Amazon DynamoDB table. Again, you can verify this in the DynamoDB console. Notice that you don't need to pass in an `id` value. An `id` is generated and returned in the results. This is because the example used an `autoId()` function in a GraphQL resolver for the partition key set on your DynamoDB resources. We will cover how you can build resolvers in a different section. Take note of the returned `id` value; you will use it in the next section to retrieve data with a GraphQL query.

Using GraphQL queries to retrieve data from a DynamoDB table in the AWS AppSync console

Now that a record exists in your database, you'll get results when you run a query. A query is one of the other fundamental operations of GraphQL. It's used to parse and retrieve information from your data source. In terms of REST APIs, this is similar to the `GET` operation. The main advantage of GraphQL queries is the ability to specify your application's exact data requirements so that you fetch the relevant data at the right time.

To query your data source

1. If you haven't already done so, sign in to the AWS Management Console and open the AppSync console.
2. Choose your API from the table.
3. In the tab to the left, choose **Queries**.
4. In the **Explorer** tab to the left of the table, under `query listTodos`, expand the `getTodo` operation:

5. In the code editor, you should see the operation code:

```
query listTodos {
  getTodo(id: "") {
    description
    id
    name
    when
    where
  }
```

In `(id:"")`, fill in the value that you saved in the result from the mutation operation. In our example, this would be:

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

6. Choose **Run**, then **listTodos**. The result will appear to the right of the editor. Our example looked like this:

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

> **Note**
>
> Queries only return the fields you specify. You can deselect the fields you don't need by deleting them from the return field:

```
{
    description
    id
    name
    when
    where
  }
```

You can also uncheck the box in the **Explorer** tab next to the field you want to delete.

7. You can also try the `listTodos` operation by repeating the steps to create an entry in your data source, then repeating the query steps with the `listTodos` operation. Here's an example where we added a second task:

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

By calling the `listTodos` operation, it returned both the old and new entries:

```
{
  "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",
```

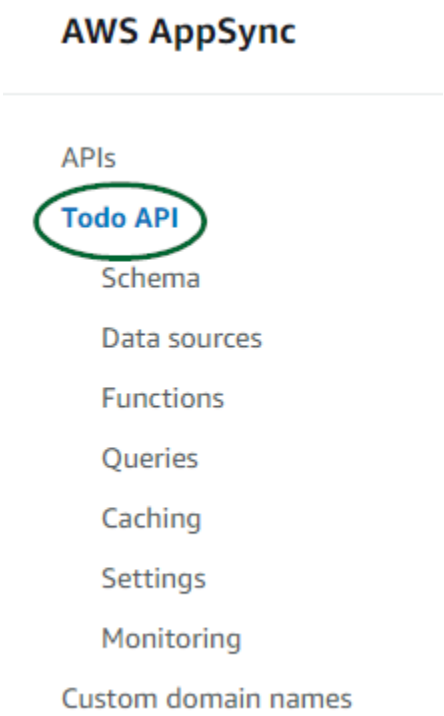
```
          "where": "Home",
          "description": "I need to mow the lawn"
        }
      ]
    }
  }
}
```

Supplemental sections for the AWS AppSync console

These sections are a reference for more advanced AWS AppSync topics. We recommend following the *Supplemental reading* section before doing anything else.

Integration

In the console tab, if you choose the name of your API, the **Integration** page appears:



It summarizes the steps for setting up your API and outlines the next steps for building a client application. The **Integrate with your app** section provides details for using the AWS Amplify toolchain to automate the process of connecting your API with iOS, Android, and JavaScript applications through config and code generation. The Amplify toolchain provides full support for

building projects from your local workstation including GraphQL provisioning and workflows for CI/CD.

The **Client Samples** section also lists sample client applications (e.g., JavaScript, iOS, Android) for testing an end-to-end experience. You can clone and download these samples, and the configuration file has the necessary information (such as your endpoint URL) you need to get started. Follow the instructions on the AWS Amplify toolchain page to run your app.

Supplemental reading

- *Designing GraphQL APIs* - This is a comprehensive guide for creating your GraphQL using a blank schema with no data sources or resolvers.

Designing GraphQL APIs with AWS AppSync

AWS AppSync allows you to create GraphQL APIs using the console experience. You caught a glimpse of this in the Launching a sample schema section. However, that guide didn't show the entire catalog of options and configurations that you could leverage in AWS AppSync.

When you choose to create a GraphQL API in the console, there are several options to explore. If you followed our Launching a sample schema guide, we showed you how to create an API from a predefined model. In the following sections, we will guide you through the rest of the options and configurations for creating GraphQL APIs in AWS AppSync.

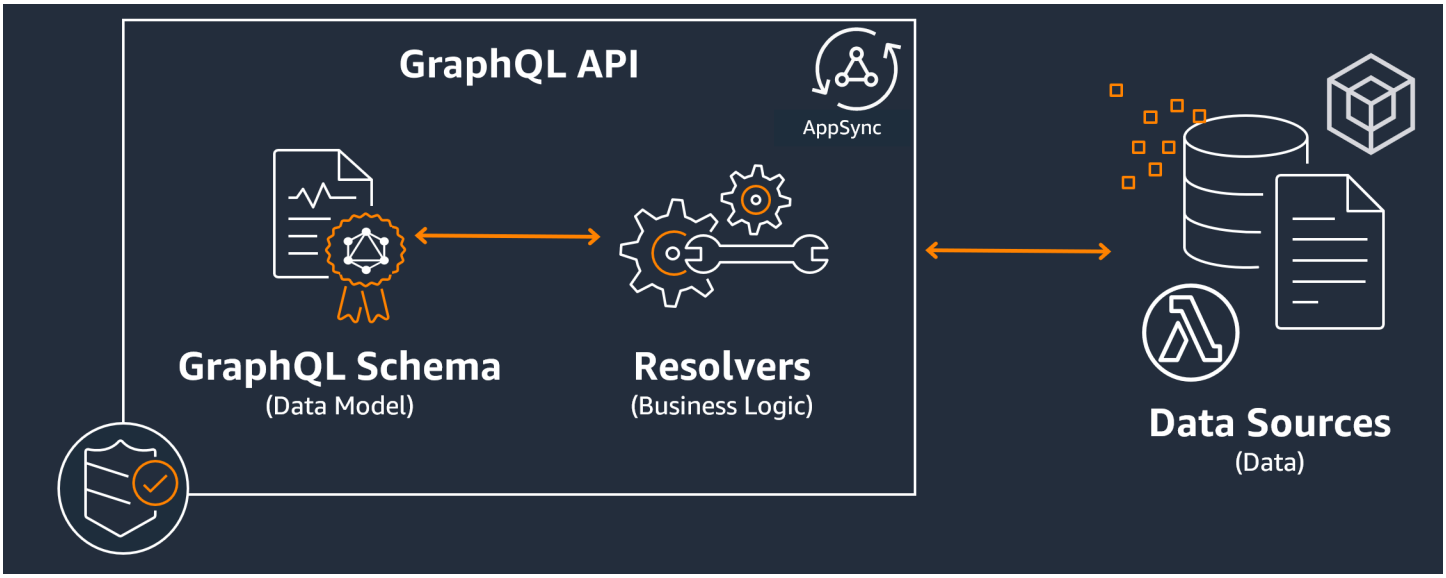
In this section, you'll review the following concepts:

1. Blank APIs or imports: This guide will run through the entire creation process for creating a GraphQL API. You'll learn how to create a GraphQL from a blank template with no model, configure data sources for your schema, and add your first resolver to a field.
2. Real-time data: This guide will show you the potential options for creating an API using AWS AppSync's WebSocket engine.
3. Merged APIs: This guide will show you how to create new GraphQL APIs by associating and merging data from multiple existing GraphQL APIs.
4. the section called “Building GraphQL APIs with RDS introspection”: This guide will show you how to integrate your Amazon RDS tables using a Data API.

Structuring a GraphQL API (blank or imported APIs)

Before you create your GraphQL API from a blank template, it would help to review the concepts surrounding GraphQL. There are three fundamental components of a GraphQL API:

1. The **schema** is the file containing the shape and definition of your data. When a request is made by a client to your GraphQL service, the data returned will follow the specification of the schema. For more information, see GraphQL schemas.
2. The **data source** is attached to your schema. When a request is made, this is where the data is retrieved and modified. For more information, see Data sources.
3. The **resolver** sits between the schema and the data source. When a request is made, the resolver performs the operation on the data from the source, then returns the result as a response. For more information, see Resolvers.



AWS AppSync manages your APIs by allowing you to create, edit, and store the code for your schemas and resolvers. Your data sources will come from external repositories such as databases, DynamoDB tables, and Lambda functions. If you're using an AWS service to store your data or are planning on doing so, AWS AppSync provides a near-seamless experience when associating data from your AWS accounts to your GraphQL APIs.

In the next section, you will learn how to create each of these components using the AWS AppSync service.

Topics

- Designing your GraphQL schema
- Attaching a data source in AWS AppSync
- Configuring resolvers in AWS AppSync
- Using an AWS AppSync API with the AWS CDK

Designing your GraphQL schema

The GraphQL schema is the foundation of any GraphQL server implementation. Each GraphQL API is defined by a **single** schema that contains types and fields describing how the data from requests will be populated. The data flowing through your API and the operations performed must be validated against the schema.

In general, the GraphQL type system describes the capabilities of a GraphQL server and is used to determine if a query is valid. A server's type system is often referred to as that server's schema

and can consist of different object types, scalar types, input types, and more. GraphQL is both declarative and strongly typed, meaning the types will be well-defined at runtime and will only return what was specified.

AWS AppSync allows you to define and configure GraphQL schemas. The following section describes how to create GraphQL schemas from scratch using AWS AppSync's services.

Structuring a GraphQL Schema

> **Tip**
>
> We recommend reviewing the Schemas section before continuing.

GraphQL is a powerful tool for implementing API services. According to GraphQL's website, GraphQL is the following:

"*GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.*"

This section covers the very first part of your GraphQL implementation, the schema. Using the quote above, a schema plays the role of "providing a complete and understandable description of the data in your API". In other words, a GraphQL schema is a textual representation of your service's data, operations, and the relations between them. The schema is considered the main entry point for your GraphQL service implementation. Unsurprisingly, it's often one of the first things you make in your project. We recommend reviewing the Schemas section before continuing.

To quote the Schemas section, GraphQL schemas are written in the *Schema Definition Language* (SDL). SDL is composed of types and fields with an established structure:

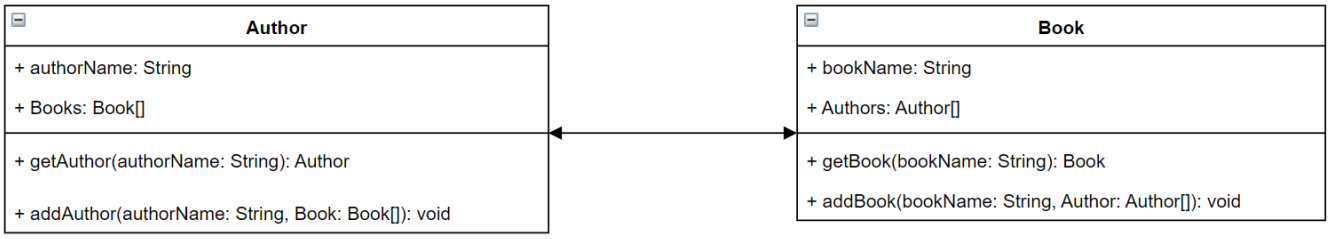
- **Types**: Types are how GraphQL defines the shape and behavior of the data. GraphQL supports a multitude of types that will be explained later in this section. Each type that's defined in your schema will contain its own scope. Inside the scope will be one or more fields that can contain a value or logic that will be used in your GraphQL service. Types fill many different roles, the most common being objects or scalars (primitive value types).
- **Fields**: Fields exist within the scope of a type and hold the value that's requested from the GraphQL service. These are very similar to variables in other programming languages. The shape

of the data you define in your fields will determine how the data is structured in a request/response operation. This allows developers to predict what will be returned without knowing how the backend of the service is implemented.

The simplest schemas will contain three different data categories:

1. **Schema roots**: Roots define the entry points of your schema. It points to the fields that will be performing some operation on the data like adding, deleting, or modifying something.
2. **Types**: These are base types that are used to represent the shape of the data. You can almost think of these as objects or abstract representations of something with defined characteristics. For example, you could make a `Person` object that represents a person in a database. Each person's characteristics will be defined inside the `Person` as fields. They can be anything like the person's name, age, job, address, etc.
3. **Special object types**: These are the types that define the behavior of the operations in your schema. Each special object type is defined once per schema. They are first placed in the schema root, then defined in the schema body. Each field in a special object type defines a single operation to be implemented by your resolver.

To put this into perspective, imagine you're creating a service that stores authors and the books they've written. Each author has a name and an array of books they've authored. Each book has a name and a list of associated authors. We also want the ability to add or retrieve books and authors. A simple UML representation of this relationship may look like this:



In GraphQL, the entities `Author` and `Book` represent two different object types in your schema:

```
type Author {
}

type Book {
```

```
}
```

`Author` contains `authorName` and `Books`, while `Book` contains `bookName` and `Authors`. These can be represented as the fields within the scope of your types:

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

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

As you can see, the type representations are very close to the diagram. However, the methods are where it gets a bit trickier. These will be placed in one of a few special object types as a field. Their special object categorization depends on their behavior. GraphQL contains three fundamental special object types: queries, mutations, and subscriptions. For more information, see Special objects.

Because `getAuthor` and `getBook` are both requesting data, they will be placed in a `Query` special object type:

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

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

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

The operations are linked to the query, which itself is linked to the schema. Adding a schema root will define the special object type (`Query` in this case) as one of your entry points. This can be done using the `schema` keyword:

```
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
}
```

Looking at the final two methods, `addAuthor` and `addBook` are adding data to your database, so they will be defined in a `Mutation` special object type. However, from the Types page, we also know that inputs directly referencing Objects aren't allowed because they're strictly output types. In this case, we can't use `Author` or `Book`, so we need to make an input type with the same fields. In this example, we added `AuthorInput` and `BookInput`, both of which accept the same fields of their respective types. Then, we create our mutation using the inputs as our parameters:

```
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
}
```

Let's review what we just did:

1. We created a schema with the Book and `Author` types to represent our entities.
2. We added the fields containing the characteristics of our entities.
3. We added a query to retrieve this information from the database.
4. We added a mutation to manipulate data in the database.
5. We added input types to replace our object parameters in the mutation to comply with GraphQL's rules.
6. We added the query and mutation to our root schema so that the GraphQL implementation understands the root type location.

As you can see, the process of creating a schema takes a lot of concepts from data modeling (especially database modeling) in general. You can think of the schema as fitting the shape of the data from the source. It also serves as the model that the resolver will implement. In the following, sections, you'll learn how to make a schema using various AWS-backed tools and services.

> **Note**
>
> The examples in the following sections are not meant to run in a real application. They are only there to showcase the commands so you can build your own applications.

Creating schemas

Your schema will be in a file called `schema.graphql`. AWS AppSync allows users to create new schemas for their GraphQL APIs using various methods. In this example, we'll be creating a blank API along with a blank schema.

Console

1. Sign in to the AWS Management Console and open the AppSync console.

   a. In the **Dashboard**, choose **Create API**.

   b. Under **API options**, choose **GraphQL APIs**, **Design from scratch**, then **Next**.

      i. For **API name**, change the prepopulated name to what your application needs.

      ii. For **contact details**, you can enter a point of contact to identify a manager for the API. This is an optional field.

      iii. Under **Private API configuration**, you can enable private API features. A private API can only be accessed from a configured VPC endpoint (VPCE). For more information, see Private APIs.

      We don't recommend enabling this feature for this example. Choose **Next** after reviewing your inputs.

   c. Under **Create a GraphQL type**, you can choose to create a DynamoDB table to use as a data source or skip this and do it later.

      For this example, choose **Create GraphQL resources later**. We will be creating a resource in a separate section.

   d. Review your inputs, then choose **Create API**.

2. You will be in the dashboard of your specific API. You can tell because the API's name will be at the top of the dashboard. If this isn't the case, you can select **APIs** in the **Sidebar**, then choose your API in the **APIs dashboard**.

- In the **Sidebar** underneath your API's name, choose **Schema**.

3. In the **Schema editor**, you can configure your `schema.graphql` file. It may be empty or filled with types generated from a model. On the right, you have the **Resolvers** section for attaching resolvers to your schema fields. We won't be looking at resolvers in this section.

CLI

> **Note**
>
> When using the CLI, make sure you have the correct permissions to access and create resources in the service. You may want to set least-privilege policies for non-admin users who need to access the service. For more information about AWS AppSync policies, see Identity and access management for AWS AppSync.
> Additionally, we recommend reading the console version first if you haven't done so already.

1. If you haven't already done so, install the AWS CLI, then add your configuration.
2. Create a GraphQL API object by running the `create-graphql-api` command.

   You'll need to type in two parameters for this particular command:

   1. The `name` of your API.
   2. The `authentication-type`, or the type of credentials used to access the API (IAM, OIDC, etc.).

   > **Note**
   >
   > Other parameters such as `Region` must be configured but will usually default to your CLI configuration values.

   An example command may look like this:

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

An output will be returned in the CLI. Here's an example:

```
{
    "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**

   This is an optional command that takes an existing schema and uploads it to the AWS AppSync service using a base-64 blob. We will not be using this command for the sake of this example.

Run the `start-schema-creation` command.

You'll need to type in two parameters for this particular command:

1. Your `api-id` from the previous step.
2. The schema `definition` is a base-64 encoded binary blob.

An example command may look like this:

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

An output will be returned:

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

This command will not return the final output after processing. You must use a separate command, `get-schema-creation-status`, to see the result. Note that these two commands are asynchronous, so you can check the output status even while the schema is still being created.

CDK

> **Tip**
>
> Before you use the CDK, we recommend reviewing the CDK's official documentation along with AWS AppSync's CDK reference.
> The steps listed below will only show a general example of the snippet used to add a particular resource. This is **not** meant to be a working solution in your production code. We also assume you already have a working app.

1. The starting point for the CDK is a bit different. Ideally, your `schema.graphql` file should already be created. You just need to create a new file with the `.graphql` file extension. This can be an empty file.
2. In general, you may have to add the import directive to the service you're using. For example, it may follow the forms:

```
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'
```

To add a GraphQL API, your stack file needs to import the AWS AppSync service:

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

> **Note**
>
> This means we're importing the entire service under the `appsync` keyword. To use this in your app, your AWS AppSync constructs will use the format `appsync.construct_name`. For instance, if we wanted to make a GraphQL API, we would say `new appsync.GraphqlApi(args_go_here)`. The following step depicts this.

3. The most basic GraphQL API will include a `name` for the API and the `schema` path.

```
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**
>
> Let's review what this snippet does. Inside the scope of `api`, we're creating a new GraphQL API by calling `appsync.GraphqlApi(scope: Construct, id: string, props: GraphqlApiProps)`. The scope is `this`, which refers to the current object. The id is *API_ID*, which will be your GraphQL API's resource name in AWS CloudFormation when it's created. The `GraphqlApiProps` contains the `name` of your GraphQL API and the `schema`. The `schema` will generate a schema (`SchemaFile.fromAsset`) by searching the absolute path (`__dirname`) for the `.graphql` file (*schema_name.graphql*). In a real scenario, your schema file will probably be inside the CDK app.
>
> To use changes made to your GraphQL API, you'll have to redeploy the app.

Adding types to schemas

Now that you've added your schema, you can start adding both your input and output types. Note that the types here shouldn't be used in real code; they're just examples to help you understand the process.

First, we'll create an object type. In real code, you don't have to start with these types. You can make any type you want at any time so long as you follow GraphQL's rules and syntax.

> **Note**
>
> These next few sections will be using the **schema editor**, so keep this open.

Console

- You can create an object type using the `type` keyword along with the type's name:

  ```
  type Type_Name_Goes_Here {}
  ```

  Inside the type's scope, you can add fields that represent the object's characteristics:

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

  Here's an example:

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

  > **Note**
  >
  > In this step, we added a generic object type with a required `id` field stored as `ID`, a `title` field stored as a `String`, and a `date` field stored as an `AWSDateTime`. To see a list of types and fields and what they do, see Schemas. To see a list of scalars and what they do, see the Type reference.

CLI

> **Note**
>
> We recommend reading the console version first if you haven't done so already.

- You can create an object type by running the `create-type` command.

  You'll need to enter a few parameters for this particular command:

  1. The `api-id` of your API.
  2. The `definition`, or the content of your type. In the console example, this was:

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

  3. The `format` of your input. In this example, we're using SDL.

  An example command may look like this:

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

  An output will be returned in the CLI. Here's an example:

  ```
  {
      "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**
>
> In this step, we added a generic object type with a required `id` field stored as `ID`, a `title` field stored as a `String`, and a `date` field stored as an `AWSDateTime`. To see a list of types and fields and what they do, see Schemas. To see a list of scalars and what they do, see Type reference.
> On a further note, you may have realized that entering the definition directly works for smaller types but is infeasible for adding larger or multiple types. You can opt to add everything in a `.graphql` file and then pass it as the input.

CDK

> **Tip**
>
> Before you use the CDK, we recommend reviewing the CDK's official documentation along with AWS AppSync's CDK reference.
> The steps listed below will only show a general example of the snippet used to add a particular resource. This is **not** meant to be a working solution in your production code. We also assume you already have a working app.

To add a type, you need to add it to your `.graphql` file. For instance, the console example was:

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

You can add your types directly to the schema like any other file.

> **Note**
>
> To use changes made to your GraphQL API, you'll have to redeploy the app.

The object type has fields that are scalar types such as strings and integers. AWS AppSync also allows you to use enhanced scalar types like `AWSDateTime` in addition to the base GraphQL scalars. Also, any field that ends in an exclamation point is required.

The `ID` scalar type in particular is a unique identifier that can be either `String` or `Int`. You can control these in your resolver code for automatic assignment.

There are similarities between special object types like `Query` and "regular" object types like the example above in that they both use the `type` keyword and are considered objects. However, for the special object types (`Query`, `Mutation`, and `Subscription`), their behavior is vastly different because they are exposed as the entry points for your API. They're also more about shaping operations rather than data. For more information, see The query and mutation types.

On the topic of special object types, the next step could be to add one or more of them to perform operations on the shaped data. In a real scenario, every GraphQL schema must at least have a root query type for requesting data. You can think of the query as one of the entry points (or endpoints) for your GraphQL server. Let's add a query as an example.

Console

- To create a query, you can simply add it to the schema file like any other type. A query would require a `Query` type and an entry in the root like this:

```
schema {
  query: Name_of_Query
}

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

  Note that *`Name_of_Query`* in a production environment will simply be called `Query` in most cases. We recommend keeping it at this value. Inside the query type, you can add fields. Each field will perform an operation in the request. As a result, most, if not all, of these fields will be attached to a resolver. However, we're not concerned with that in this section. Regarding the format of the field operation, it might look like this:

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

Here's an example:

```
schema {
  query: Query
}

type Query {
  getObj: [Obj_Type_1]
}

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

> **Note**
>
> In this step, we added a `Query` type and defined it in our `schema` root. Our `Query` type defined a `getObj` field that returns a list of `Obj_Type_1` objects. Note that `Obj_Type_1` is the object of the previous step. In production code, your field operations will normally be working with data shaped by objects like `Obj_Type_1`. In addition, fields like `getObj` will normally have a resolver to perform the business logic. That will be covered in a different section.
> As an additional note, AWS AppSync automatically adds a schema root during exports, so technically you don't have to add it directly to the schema. Our service will automatically process duplicate schemas. We're adding it here as a best practice.

CLI

> **Note**
>
> We recommend reading the console version first if you haven't done so already.

1. Create a `schema` root with a `query` definition by running the `create-type` command.

You'll need to enter a few parameters for this particular command:

1. The `api-id` of your API.
2. The `definition`, or the content of your type. In the console example, this was:

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

3. The `format` of your input. In this example, we're using SDL.

An example command may look like this:

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

An output will be returned in the CLI. Here's an example:

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

Note

Note that if you didn't input something correctly in the `create-type` command, you can update your schema root (or any type in the schema) by running the `update-type` command. In this example, we'll be temporarily changing the schema root to contain a `subscription` definition.
You'll need to enter a few parameters for this particular command:

1. The `api-id` of your API.
2. The `type-name` of your type. In the console example, this was `schema`.

3. The `definition`, or the content of your type. In the console example, this was:

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

The schema after adding a `subscription` will look like this:

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

4. The `format` of your input. In this example, we're using SDL.

An example command may look like this:

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

An output will be returned in the CLI. Here's an example:

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

Adding preformatted files will still work in this example.

2. Create a `Query` type by running the `create-type` command.

You'll need to enter a few parameters for this particular command:

1. The `api-id` of your API.
2. The `definition`, or the content of your type. In the console example, this was:

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

3. The `format` of your input. In this example, we're using SDL.

An example command may look like this:

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

An output will be returned in the CLI. Here's an example:

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

Note

In this step, we added a `Query` type and defined it in your `schema` root. Our `Query` type defined a `getObj` field that returned a list of `Obj_Type_1` objects.
In the `schema` root code `query: Query`, the `query:` part indicates that a query was defined in your schema, while the `Query` part indicates the actual special object name.

CDK

Tip

Before you use the CDK, we recommend reviewing the CDK's official documentation along with AWS AppSync's CDK reference.

> The steps listed below will only show a general example of the snippet used to add a particular resource. This is **not** meant to be a working solution in your production code. We also assume you already have a working app.

You'll need to add your query and the schema root to the `.graphql` file. Our example looked like the example below, but you'll want to replace it with your actual schema code:

```
schema {
  query: Query
}

type Query {
  getObj: [Obj_Type_1]
}

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

You can add your types directly to the schema like any other file.

> **Note**
>
> Updating the schema root is optional. We added it to this example as a best practice. To use changes made to your GraphQL API, you'll have to redeploy the app.

You've now seen an example of creating both objects and special objects (queries). You've also seen how these can be interconnected to describe data and operations. You can have schemas with only the data description and one or more queries. However, we'd like to add another operation to add data to the data source. We'll add another special object type called `Mutation` that modifies data.

Console

- A mutation will be called `Mutation`. Like `Query`, the field operations inside `Mutation` will describe an operation and will be attached to a resolver. Also, note that we need to define it in the `schema` root because it's a special object type. Here's an example of a mutation:

```
schema {
  mutation: Name_of_Mutation
}

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

A typical mutation will be listed in the root like a query. The mutation is defined using the `type` keyword along with the name. *`Name_of_Mutation`* will usually be called `Mutation`, so we recommend keeping it that way. Each field will also perform an operation. Regarding the format of the field operation, it might look like this:

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

Here's an example:

```
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

In this step, we added a `Mutation` type with an `addObj` field. Let's summarize what this field does:

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

`addObj` is using the `Obj_Type_1` object to perform an operation. This is apparent due to the fields, but the syntax proves this in the `: Obj_Type_1` return type. Inside `addObj`, it's accepting the `id`, `title`, and `date` fields from the `Obj_Type_1` object as parameters. As you may see, it looks a lot like a method declaration. However, we haven't described the behavior of our method yet. As stated earlier, the schema is only there to define what the data and operations will be and not how they operate. Implementing the actual business logic will come later when we create our first resolvers.
Once you're done with your schema, there's an option to export it as a `schema.graphql` file. In the **Schema editor**, you can choose **Export schema** to download the file in a supported format.
As an additional note, AWS AppSync automatically adds a schema root during exports, so technically you don't have to add it directly to the schema. Our service will automatically process duplicate schemas. We're adding it here as a best practice.

CLI

Note

We recommend reading the console version first if you haven't done so already.

1. Update your root schema by running the `update-type` command.

   You'll need to enter a few parameters for this particular command:

   1. The `api-id` of your API.
   2. The `type-name` of your type. In the console example, this was `schema`.

3. The `definition`, or the content of your type. In the console example, this was:

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

4. The `format` of your input. In this example, we're using SDL.

An example command may look like this:

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

An output will be returned in the CLI. Here's an example:

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

2. Create a `Mutation` type by running the `create-type` command.

You'll need to enter a few parameters for this particular command:

1. The `api-id` of your API.
2. The `definition`, or the content of your type. In the console example, this was

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

3. The `format` of your input. In this example, we're using SDL.

An example command may look like this:

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

An output will be returned in the CLI. Here's an example:

```
{
    "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**
>
> Before you use the CDK, we recommend reviewing the CDK's official documentation along with AWS AppSync's CDK reference.
> The steps listed below will only show a general example of the snippet used to add a particular resource. This is **not** meant to be a working solution in your production code. We also assume you already have a working app.

You'll need to add your query and the schema root to the `.graphql` file. Our example looked like the example below, but you'll want to replace it with your actual schema code:

```
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**
>
> Updating the schema root is optional. We added it to this example as a best practice. To use changes made to your GraphQL API, you'll have to redeploy the app.

Optional considerations - Using enums as statuses

At this point, you know how to make a basic schema. However, there are many things you could add to increase the schema's functionality. One common thing found in applications is the use of enums as statuses. You can use an enum to force a specific value from a set of values to be chosen when called. This is good for things that you know will not change drastically over long periods of time. Hypothetically speaking, we could add an enum that returns the status code or String in the response.

As an example, let's assume we're making a social media app that's storing a user's post data in the backend. Our schema contains a `Post` type that represents an individual post's data:

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

Our `Post` will contain a unique `id`, post `title`, `date` of posting, and an enum called `PostStatus` that represents the post's state as it's processed by the app. For our operations, we'll have a query that returns all post data:

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

We'll also have a mutation that adds posts to the data source:

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

Looking at our schema, the `PostStatus` enum could have several statuses. We might want the three basic states called `success` (post successfully processed), `pending` (post being processed), and `error` (post unable to be processed). To add the enum, we could do this:

```
enum PostStatus {
  success
  pending
  error
}
```

The full schema might look like this:

```
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
}
```

If a user adds a `Post` in the application, the `addPost` operation will be called to process that data. As the resolver attached to `addPost` processes the data, it will continually update the `poststatus` with the status of the operation. When queried, the `Post` will contain the final status of the data. Keep in mind, we're only describing how we want the data to work in the schema. We're assuming a lot about the implementation of our resolver(s), which will implement the actual business logic for handling the data to fulfill the request.

Optional considerations - Subscriptions

Subscriptions in AWS AppSync are invoked as a response to a mutation. You configure this with a `Subscription` type and `@aws_subscribe()` directive in the schema to denote which mutations invoke one or more subscriptions. For more information about configuring subscriptions, see Real-time data.

Optional considerations - Relations and pagination

Suppose you had a million `Posts` stored in a DynamoDB table, and you wanted to return some of that data. However, the example query given above only returns all posts. You wouldn't want to fetch all of these every time you made a request. Instead, you would want to paginate through them. Make the following changes to your schema:

- In the `getPosts` field, add two input arguments: `nextToken` (iterator) and `limit` (iteration limit).
- Add a new `PostIterator` type containing `Posts` (retrieves the list of `Post` objects) and `nextToken` (iterator) fields.
- Change `getPosts` so that it returns `PostIterator` and not a list of `Post` objects.

```
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
}
```

The `PostIterator` type allows you to return a portion of the list of `Post` objects and a `nextToken` for getting the next portion. Inside `PostIterator`, there is a list of `Post` items (`[Post]`) that is returned with a pagination token (`nextToken`). In AWS AppSync, this would be connected to Amazon DynamoDB through a resolver and automatically generated as an encrypted token. This converts the value of the `limit` argument to the `maxResults` parameter and the `nextToken` argument to the `exclusiveStartKey` parameter. For examples and the built-in template samples in the AWS AppSync console, see Resolver reference (JavaScript).

Attaching a data source in AWS AppSync

Data sources are resources in your AWS account that GraphQL APIs can interact with. AWS AppSync supports a multitude of data sources like AWS Lambda, Amazon DynamoDB, relational databases (Amazon Aurora Serverless), Amazon OpenSearch Service, and HTTP endpoints. An AWS AppSync API can be configured to interact with multiple data sources, enabling you to aggregate data in

a single location. AWS AppSync can use existing AWS resources from your account or provision DynamoDB tables on your behalf from a schema definition.

The following section will show you how to attach a data source to your GraphQL API.

Types of data sources

Now that you have created a schema in the AWS AppSync console, you can attach a data source to it. When you initially create an API, there's an option to provision an Amazon DynamoDB table during the creation of the predefined schema. However, we won't be covering that option in this section. You can see an example of this in the Launching a schema section.

Instead, we'll be looking at all of the data sources AWS AppSync supports. There are many factors that go into picking the right solution for your application. The sections below will provide some additional context for each data source. For general information about data sources, see Data sources.

Amazon DynamoDB

Amazon DynamoDB is one of AWS' main storage solutions for scalable applications. The core component of DynamoDB is the **table**, which is simply a collection of data. You will typically create tables based on entities like `Book` or `Author`. Table entry information is stored as **items**, which are groups of fields that are unique to each entry. A full item represents a row/record in the database. For example, an item for a `Book` entry might include `title` and `author` along with their values. The individual fields like the `title` and `author` are called **attributes**, which are akin to column values in relational databases.

As you can guess, tables will be used to store data from your application. AWS AppSync allows you to hook up your DynamoDB tables to your GraphQL API to manipulate data. Take this use case from the *Front-end web and mobile blog*. This application lets users sign up for a social media app. Users can join groups and upload posts that are broadcasted to other users subscribed to the group. Their application stores user, post, and user group information in DynamoDB. The GraphQL API (managed by AWS AppSync) interfaces with the DynamoDB table. When a user makes a change in the system that will be reflected on the front-end, the GraphQL API retrieves these changes and broadcasts them to other users in real time.

AWS Lambda

Lambda is an event-driven service that automatically builds the necessary resources to run code as a response to an event. Lambda uses **functions**, which are group statements containing the code,

dependencies, and configurations for executing a resource. Functions automatically execute when they detect a **trigger**, a group of activities that invoke your function. A trigger could be anything like an application making an API call, an AWS service in your account spinning up a resource, etc. When triggered, functions will process **events**, which are JSON documents containing the data to modify.

Lambda is good for running code without having to provision the resources to run it. Take this use case from the *Front-end web and mobile blog*. This use case is a bit similar to the one showcased in the DynamoDB section. In this application, the GraphQL API is responsible for defining the operations for things like adding posts (mutations) and fetching that data (queries). To implement the functionality of their operations (e.g., `getPost ( id: String ! ) : Post`, `getPostsByAuthor ( author: String ! ) : [ Post ]`), they use Lambda functions to process inbound requests. Under *Option 2: AWS AppSync with Lambda resolver*, they use the AWS AppSync service to maintain their schema and link a Lambda data source to one of the operations. When the operation is called, Lambda interfaces with the Amazon RDS proxy to perform the business logic on the database.

Amazon RDS

Amazon RDS lets you quickly build and configure relational databases. In Amazon RDS, you'll create a generic **database instance** that will serve as the isolated database environment in the cloud. In this instance, you'll use a **DB engine**, which is the actual RDBMS software (PostgreSQL, MySQL, etc.). The service offloads much of the backend work by providing scalability using AWS' infrastructure, security services such as patching and encryption, and lowered administrative costs for deployments.

Take the same use case from the Lambda section. Under *Option 3: AWS AppSync with Amazon RDS resolver*, another option presented is linking the GraphQL API in AWS AppSync to Amazon RDS directly. Using a data API, they associate the database with the GraphQL API. A resolver is attached to a field (usually a query, mutation, or subscription) and implements the SQL statements needed to access the database. When a request calling the field is made by the client, the resolver executes the statements and returns the response.

Amazon EventBridge

In EventBridge, you'll create **event buses**, which are pipelines that receive events from services or applications you attach (the **event source**) and process them based on a set of rules. An **event** is some state change in an execution environment, while a **rule** is a set of filters for events. A rule follows an **event pattern**, or metadata of an event's state change (id, Region, account number,

ARN(s), etc.). When an event matches the event pattern, EventBridge will send the event across the pipeline to the destination service (**target**) and trigger the action specified in the rule.

EventBridge is good for routing state-changing operations to some other service. Take this use case from the *Front-end web and mobile blog*. The example depicts an e-commerce solution that has several teams maintaining different services. One of these services provides order updates to the customer at each step of the delivery (order placed, in progress, shipped, delivered, etc.) on the front-end. However, the front-end team managing this service doesn't have direct access to the ordering system data as that's maintained by a separate backend team. The backend team's ordering system is also described as a black box, so it's hard to glean information about the way they're structuring their data. However, the backend team did set up a system that published order data through an event bus managed by EventBridge. To access the data coming from the event bus and route it to the front-end, the front-end team created a new target pointing to their GraphQL API sitting in AWS AppSync. They also created a rule to only send data relevant to the order update. When an update is made, the data from the event bus is sent to the GraphQL API. The schema in the API processes the data, then passes it to the front-end.

None data sources

If you aren't planning on using a data source, you can set it to none. A none data source, while still explicitly categorized as a data source, isn't a storage medium. Typically, a resolver will invoke one or more data sources at some point to process the request. However, there are situations where you may not need to manipulate a data source. Setting the data source to none will run the request, skip the data invocation step, then run the response.

Take the same use case from the EventBridge section. In the schema, the mutation processes the status update, then sends it out to subscribers. Recalling how resolvers work, there's usually at least one data source invocation. However, the data in this scenario was already sent automatically by the event bus. This means there's no need for the mutation to perform a data source invocation; the order status can simply be handled locally. The mutation is set to none, which acts as a pass-through value with no data source invocation. The schema is then populated with the data, which is sent out to subscribers.

OpenSearch

Amazon OpenSearch Service is a suite of tools to implement full-text searching, data visualization, and logging. You can use this service to query the structured data you've uploaded.

In this service, you'll create instances of OpenSearch. These are called **nodes**. In a node, you'll be adding at least one **index**. Indices conceptually are a bit like tables in relational databases.

(However, OpenSearch isn't ACID compliant, so it shouldn't be used that way). You'll populate your index with data that you upload to the OpenSearch service. When your data is uploaded, it will be indexed in one or more shards that exist in the index. A **shard** is like a partition of your index that contains some of your data and can be queried separately from other shards. Once uploaded, your data will be structured as JSON files called **documents**. You can then query the node for data in the document.

HTTP endpoints

You can use HTTP endpoints as data sources. AWS AppSync can send requests to the endpoints with the relevant information like params and payload. The HTTP response will be exposed to the resolver, which will return the final response after it finishes its operation(s).

Adding a data source

If you created a data source, you can link it to the AWS AppSync service and, more specifically, the API.

Console

1. Sign in to the AWS Management Console and open the AppSync console.
   a. Choose your API in the **Dashboard**.
   b. In the **Sidebar**, choose **Data Sources**.
2. Choose **Create data source**.
   a. Give your data source a name. You can also give it a description, but that's optional.
   b. Choose your **Data source type**.
   c. For DynamoDB, you'll have to choose your Region, then the table in the Region. You can dictate interaction rules with your table by choosing to make a new generic table role or importing an existing role for the table. You can enable versioning, which can automatically create versions of data for each request when multiple clients are trying to update data at the same time. Versioning is used to keep and maintain multiple variants of data for conflict detection and resolution purposes. You can also enable automatic schema generation, which takes your data source and generates some of the CRUD, `List`, and `Query` operations needed to access it in your schema.

For OpenSearch, you'll have to choose your Region, then the domain (cluster) in the Region. You can dictate interaction rules with your domain by choosing to make a new generic table role or importing an existing role for the table.

For Lambda, you'll have to choose your Region, then the ARN of the Lambda function in the Region. You can dictate interaction rules with your Lambda function by choosing to make a new generic table role or importing an existing role for the table.

For HTTP, you'll have to enter your HTTP endpoint.

For EventBridge, you'll have to choose your Region, then the event bus in the Region. You can dictate interaction rules with your event bus by choosing to make a new generic table role or importing an existing role for the table.

For RDS, you'll have to choose your Region, then the secret store (username and password), database name, and schema.

For none, you will add a data source with no actual data source. This is for handling resolvers locally rather than through an actual data source.

> **Note**
>
> If you're importing existing roles, they need a trust policy. For more information, see the IAM trust policy.

3. Choose **Create**.

> **Note**
>
> Alternatively, if you're creating a DynamoDB data source, you can go to the **Schema** page in the console, choose **Create Resources** at the top of the page, then fill out a predefined model to convert into a table. In this option, you will fill out or import the base type, configure the basic table data including the partition key, and review the schema changes.

CLI

- Create your data source by running the `create-data-source` command.

You'll need to enter a few parameters for this particular command:

1. The `api-id` of your API.
2. The `name` of your table.
3. The `type` of data source. Depending on the data source type you choose, you may need to enter a `service-role-arn` and a `-config` tag.

An example command may look like this:

```
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**
>
> Before you use the CDK, we recommend reviewing the CDK's official documentation along with AWS AppSync's CDK reference.
> The steps listed below will only show a general example of the snippet used to add a particular resource. This is **not** meant to be a working solution in your production code. We also assume you already have a working app.

To add your particular data source, you'll need to add the construct to your stack file. A list of data source types can be found here:

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

1. In general, you may have to add the import directive to the service you're using. For example, it may follow the forms:

```
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'
```

   For example, here's how you could import the AWS AppSync and DynamoDB services:

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

2. Some services like RDS require some additional setup in the stack file before creating the data source (e.g., VPC creation, roles, and access credentials). Consult the examples in the relevant CDK pages for more information.

3. For most data sources, especially AWS services, you'll be creating a new instance of the data source in your stack file. Typically, this will look like the following:

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

   For example, here's an example Amazon DynamoDB table:

```
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**
>
> Most data sources will have at least one required prop (will be denoted **without** a ? symbol). Consult the CDK documentation to see which props are needed.

4. Next, you need to link the data source to the GraphQL API. The recommended method is to add it when you make a function for your pipeline resolver. For instance, the snippet below is a function that scans all elements in a DynamoDB table:

```
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,
});
```

In the `dataSource` props, you can call the GraphQL API (`add_api`) and use one of its built-in methods (`addDynamoDbDataSource`) to make the association between the table and the GraphQL API. The arguments are the name of this link that will exist in the AWS AppSync console (`data_source_name_in_console` in this example) and the table method (`add_ddb_table`). More on this topic will be revealed in the next section when you start making resolvers.

There are alternative methods for linking a data source. You could technically add `api` to the props list in the table function. For example, here's the snippet from step 3 but with an `api` props containing a GraphQL API:

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

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

 ...

  api: add_api
```

```
});
```

Alternatively, you can call the `GraphqlApi` construct separately:

```
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);
```

We recommend only creating the association in the function's props. Otherwise, you'll either have to link your resolver function to the data source manually in the AWS AppSync console (if you want to keep using the console value `data_source_name_in_console`) or create a separate association in the function under another name like `data_source_name_in_console_2`. This is due to limitations in how the props process information.

> **Note**
>
> You'll have to redeploy the app to see your changes.

IAM trust policy

If you're using an existing IAM role for your data source, you need to grant that role the appropriate permissions to perform operations on your AWS resource, such as `PutItem` on an Amazon DynamoDB table. You also need to modify the trust policy on that role to allow AWS AppSync to use it for resource access as shown in the following example policy:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
        "Effect": "Allow",
        "Principal": {
            "Service": "appsync.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
        }
    ]
}
```

You can also add conditions to your trust policy to limit access to the data source as desired. Currently, `SourceArn` and `SourceAccount` keys can be used in these conditions. For example, the following policy limits access to your data source to the account `123456789012`:

JSON

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

Alternatively, you can limit access to a data source to a specific API, such as `abcdefghijklmnopq`, using the following policy:

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"
        }
      }
    }
  ]
}
```

You can limit access to all AWS AppSync APIs from a specific region, such as `us-east-1`, using the following policy:

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/*"
        }
      }
```

```
    }
  ]
}
```

In the next section (Configuring Resolvers), we'll add our resolver business logic and attach it to the fields in our schema to process the data in our data source.

For more information regarding role policy configuration, see Modifying a role in the *IAM User Guide*.

For more information regarding cross-account access of AWS Lambda resolvers for AWS AppSync, see Building cross-account AWS Lambda resolvers for AWS AppSync.

Configuring resolvers in AWS AppSync

In the previous sections, you learned how to create your GraphQL schema and data source, then linked them together in the AWS AppSync service. In your schema, you may have established one or more fields (operations) in your query and mutation. While the schema described the kinds of data the operations would request from the data source, it never implemented how those operations would behave around the data.

An operation's behavior is always implemented in the resolver, which will be linked to the field performing the operation. For more information about how resolvers work in general, see the Resolvers page.

In AWS AppSync, your resolver is tied to a runtime, which is the environment in which your resolver executes. Runtimes dictate the language that your resolver will be written in. There are currently two supported runtimes: APPSYNC_JS (JavaScript) and Apache Velocity Template Language (VTL).

When implementing resolvers, there is a general structure they follow:

- **Before step**: When a request is made by the client, the resolvers for the schema fields being used (typically your queries, mutations, subscriptions) are passed the request data. The resolver will begin processing the request data with a before step handler, which allows some preprocessing operations to be performed before the data moves through the resolver.
- **Function(s)**: After the before step runs, the request is passed to the functions list. The first function in the list will execute against the data source. A function is a subset of your resolver's code containing its own request and response handler. A request handler will take the request

data and perform operations against the data source. The response handler will process the data source's response before passing it back to the list. If there is more than one function, the request data will be sent to the next function in the list to be executed. Functions in the list will be executed serially in the order defined by the developer. Once all functions have been executed, the final result is passed to the after step.

- **After step**: The after step is a handler function that allows you to perform some final operations on the final function's response before passing it to the GraphQL response.

This flow is an example of a pipeline resolver. Pipeline resolvers are supported in both runtimes. However, this is a simplified explanation of what pipeline resolvers can do. Also, we're describing only one possible resolver configuration. For more information about supported resolver configurations, see the JavaScript resolvers overview for APPSYNC_JS or the Resolver mapping template overview for VTL.

As you can see, resolvers are modular. In order for the components of the resolver to work properly, they must be able to peer into the state of the execution from other components. From the Resolvers section, you know that each component in the resolver can be passed vital information about the state of the execution as a set of arguments (`args`, `context`, etc.). In AWS AppSync, this is handled strictly by the `context`. It's a container for the information about the field being resolved. This can include everything from arguments being passed, results, authorization data, header data, etc. For more information about the context, see the Resolver context object reference for APPSYNC_JS or the Resolver mapping template context reference for VTL.

The context isn't the only tool you can use to implement your resolver. AWS AppSync supports a wide range of utilities for value generation, error handling, parsing, conversion, etc. You can see a list of utilities here for APPSYNC_JS or here for VTL.

In the following sections, you will learn how to configure resolvers in your GraphQL API.

Topics

Creating basic queries (JavaScript)

GraphQL resolvers connect the fields in a type's schema to a data source. Resolvers are the mechanism by which requests are fulfilled.

Resolvers in AWS AppSync use JavaScript to convert a GraphQL expression into a format the data source can use. Alternatively, mapping templates can be written in Apache Velocity Template Language (VTL) to convert a GraphQL expression into a format the data source can use.

This section describes how to configure resolvers using JavaScript. The Resolver tutorials (JavaScript) section provides in-depth tutorials on how to implement resolvers using JavaScript. The Resolver reference (JavaScript) section provides an explanation of utility operations that can be used with JavaScript resolvers.

We recommend following this guide before attempting to use any of the aforementioned tutorials.

In this section, we will walk through how to create and configure resolvers for queries and mutations.

> **Note**
>
> This guide assumes you have created your schema and have at least one query or mutation. If you're looking for subscriptions (real-time data), then see this guide.

In this section, we'll provide some general steps for configuring resolvers along with an example that uses the schema below:

```
// 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]
}
```

Creating basic query resolvers

This section will show you how to make a basic query resolver.

Console

1. Sign in to the AWS Management Console and open the AppSync console.

   a. In the **APIs dashboard**, choose your GraphQL API.

   b. In the **Sidebar**, choose **Schema**.

2. Enter the details of your schema and data source. See the Designing your schema and Attaching a data source sections for more information.

3. Next to the **Schema** editor, There's a window called **Resolvers**. This box contains a list of the types and fields as defined in your **Schema** window. You can attach resolvers to fields. You will most likely be attaching resolvers to your field operations. In this section, we'll look at simple query configurations. Under the **Query** type, choose **Attach** next to your query's field.

4. On the **Attach resolver** page, under **Resolver type**, you can choose between pipeline or unit resolvers. For more information about these types, see Resolvers. This guide will make use of `pipeline resolvers`.

   > **Tip**
   >
   > When creating pipeline resolvers, your data source(s) will be attached to the pipeline function(s). Functions are created after you create the pipeline resolver itself, which is why there's no option to set it in this page. If you're using a unit resolver, the data source is tied directly to the resolver, so you would set it in this page.

   For **Resolver runtime**, choose APPSYNC_JS to enable the JavaScript runtime.

5. You can enable caching for this API. We recommend turning this feature off for now. Choose **Create**.

6. On the **Edit resolver** page, there's a code editor called **Resolver code** that allows you to implement the logic for the resolver handler and response (before and after steps). For more information, see the JavaScript resolvers overview.

> **Note**
>
> In our example, we're just going to leave the request blank and the response set to return the last data source result from the context:
>
> ```
> import {util} from '@aws-appsync/utils';
>
> export function request(ctx) {
>     return {};
> }
>
> export function response(ctx) {
>     return ctx.prev.result;
> }
> ```

Below this section, there's a table called **Functions**. Functions allow you to implement code that can be reused across multiple resolvers. Instead of constantly rewriting or copying code, you can store the source code as a function to be added to a resolver whenever you need it.

Functions make up the bulk of a pipeline's operation list. When using multiple functions in a resolver, you set the order of the functions, and they will be run in that order sequentially. They are executed after the request function runs and before the response function begins.

To add a new function, under **Functions**, choose **Add function**, then **Create new function**. Alternatively, you may see a **Create function** button to choose instead.

a. Choose a data source. This will be the data source on which the resolver acts.

> **Note**
>
> In our example, we're attaching a resolver for `getPost`, which retrieves a `Post` object by `id`. Let's assume we already set up a DynamoDB table for this schema. Its partition key is set to the `id` and is empty.

b. Enter a `Function name`.

c. Under **Function code**, you'll need to implement the function's behavior. This might be confusing, but each function will have its own local request and response handler. The request runs, then the data source invocation is made to handle the request, then the data source response is processed by the response handler. The result is stored in the context object. Afterward, the next function in the list will run or will be passed to the after step response handler if it's the last one.

> **Note**
>
> In our example, we're attaching a resolver to `getPost`, which gets a list of `Post` objects from the data source. Our request function will request the data from our table, the table will pass its response to the context (ctx), then the response will return the result in the context. AWS AppSync's strength lies in its interconnectedness with other AWS services. Because we're using DynamoDB, we have a suite of operations to simplify things like this. We have some boilerplate examples for other data source types as well.
> Our code will look like this:

```
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;
}
```

In this step, we added two functions:

- `request`: The request handler performs the retrieval operation against the data source. The argument contains the context object (`ctx`), or some data that is available to all resolvers performing a particular operation. For example, it might contain authorization data, the field names being resolved, etc. The return statement performs a `Scan` operation (see here for examples). Because we're working with DynamoDB, we're allowed to use some of the operations from that service. The scan performs a basic fetch of all items in our table. The result of this operation is stored in the context object as a `result` container before being passed to the response handler. The `request` is run before the response in the pipeline.
- `response`: The response handler that returns the output of the `request`. The argument is the updated context object, and the return statement is `ctx.prev.result`. At this point in the guide, you may not be familiar with this value. `ctx` refers to the context object. `prev` refers to the previous operation in the pipeline, which was our `request`. The `result` contains the result(s) of the resolver as it moves through the pipeline. If you put it all together, `ctx.prev.result` is returning the result of the last operation performed, which was the request handler.

d. Choose **Create** after you're done.

7. Back on the resolver screen, under **Functions**, choose the **Add function** drop-down and add your function to your functions list.
8. Choose **Save** to update the resolver.

CLI

To add your function

- Create a function for your pipeline resolver using the `create-function` command.

  You'll need to enter a few parameters for this particular command:

  1. The `api-id` of your API.

2. The `name` of the function in the AWS AppSync console.
3. The `data-source-name`, or the name of the data source the function will use. It must already be created and linked to your GraphQL API in the AWS AppSync service.
4. The `runtime`, or environment and language of the function. For JavaScript, the name must be `APPSYNC_JS`, and the runtime, `1.0.0`.
5. The `code`, or request and response handlers of your function. While you can type it in manually, it's far easier to add it to a .txt file (or a similar format) and then pass it in as the argument.

> **Note**
>
> Our query code will be in a file passed in as the argument:
>
> ```
> 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;
> }
> ```

An example command may look like this:

```
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}
```

An output will be returned in the CLI. Here's an example:

```
{
    "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**
>
> Make sure you record the `functionId` somewhere as this will be used to attach the function to the resolver.

To create your resolver

- Create a pipeline function for `Query` by running the `create-resolver` command.

  You'll need to enter a few parameters for this particular command:

  1. The `api-id` of your API.
  2. The `type-name`, or the special object type in your schema (Query, Mutation, Subscription).
  3. The `field-name`, or the field operation inside the special object type you want to attach the resolver to.
  4. The `kind`, which specifies a unit or pipeline resolver. Set this to `PIPELINE` to enable pipeline functions.

5. The `pipeline-config`, or the function(s) to attach to the resolver. Make sure you know the `functionId` values of your functions. Order of listing matters.
6. The `runtime`, which was `APPSYNC_JS` (JavaScript). The `runtimeVersion` currently is `1.0.0`.
7. The `code`, which contains the before and after step handlers.

> **Note**
>
> Our query code will be in a file passed in as the argument:
>
> ```
> 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;
> }
> ```

An example command may look like this:

```
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}
```

An output will be returned in the CLI. Here's an example:

```
{
    "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

Before you use the CDK, we recommend reviewing the CDK's official documentation along with AWS AppSync's CDK reference.
The steps listed below will only show a general example of the snippet used to add a particular resource. This is **not** meant to be a working solution in your production code. We also assume you already have a working app.

A basic app will need the following things:

1. Service import directives

2. Schema code
3. Data source generator
4. Function code
5. Resolver code

From the Designing your schema and Attaching a data source sections, we know that the stack file will include the import directives of the form:

```
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

In previous sections, we only stated how to import AWS AppSync constructs. In real code, you'll have to import more services just to run the app. In our example, if we were to create a very simple CDK app, we would at least import the AWS AppSync service along with our data source, which was a DynamoDB table. We would also need to import some additional constructs to deploy the app:

```
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';
```

To summarize each of these:

- `import * as cdk from 'aws-cdk-lib';`: This allows you to define your CDK app and constructs such as the stack. It also contains some useful utility functions for our application like manipulating metadata. If you're familiar with this import directive, but are wondering why the cdk core library is not being used here, see the Migration page.
- `import * as appsync from 'aws-cdk-lib/aws-appsync';`: This imports the AWS AppSync service.
- `import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';`: This imports the DynamoDB service.

- `import { Construct } from 'constructs';`: We need this to define the root construct.

The type of import depends on the services you're calling. We recommend looking at the CDK documentation for examples. The schema at the top of the page will be a separate file in your CDK app as a `.graphql` file. In the stack file, we can associate it with a new GraphQL using the form:

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

Note

In the scope `add_api`, we're adding a new GraphQL API using the `new` keyword followed by `appsync.GraphqlApi(scope: Construct, id: string , props: GraphqlApiProps)`. Our scope is `this`, the CFN id is `graphQL-example`, and our props are `my-first-api` (name of the API in the console) and `schema.graphql` (the absolute path to the schema file).

To add a data source, you'll first have to add your data source to the stack. Then, you need to associate it with the GraphQL API using the source-specific method. The association will happen when you make your resolver function. In the meantime, let's use an example by creating the DynamoDB table using `dynamodb.Table`:

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

> **Note**
>
> If we were to use this in our example, we'd be adding a new DynamoDB table with the CFN id of `posts-table` and a partition key of `id (S)`.

Next, we need to implement our resolver in the stack file. Here's an example of a simple query that scans for all items in a DynamoDB table:

```
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

First, we created a function called `add_func`. This order of creation may seem a bit counterintuitive, but you have to create the functions in your pipeline resolver before you make the resolver itself. A function follows the form:

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

Our scope was `this`, our CFN id was `func-get-posts`, and our props contained the actual function details. Inside props, we included:

- The `name` of the function that will be present in the AWS AppSync console (`get_posts_func_1`).
- The GraphQL API we created earlier (`add_api`).
- The data source; this is the point where we link the data source to the GraphQL API value, then attach it to the function. We take the table we created (`add_ddb_table`) and attach it to the GraphQL API (`add_api`) using one of the `GraphqlApi` methods (`addDynamoDbDataSource`). The id value (`table-for-posts`) is the name of the data source in the AWS AppSync console. For a list of source-specific methods, see the following pages:
  - DynamoDbDataSource
  - EventBridgeDataSource
  - HttpDataSource
  - LambdaDataSource
  - NoneDataSource
  - OpenSearchDataSource
  - RdsDataSource
- The code contains our function's request and response handlers, which is a simple scan and return.
- The runtime specifies that we want to use the APPSYNC_JS runtime version 1.0.0. Note that this is currently the only version available for APPSYNC_JS.

Next, we need to attach the function to the pipeline resolver. We created our resolver using the form:

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

Our scope was `this`, our CFN id was `pipeline-resolver-get-posts`, and our props contained the actual function details. Inside the props, we included:

- The GraphQL API we created earlier (`add_api`).
- The special object type name; this is a query operation, so we simply added the value `Query`.
- The field name (`getPost`) is the name of the field in the schema under the `Query` type.
- The code contains your before and after handlers. Our example just returns whatever results were in the context after the function performed its operation.
- The runtime specifies that we want to use the APPSYNC_JS runtime version 1.0.0. Note that this is currently the only version available for APPSYNC_JS.
- The pipeline config contains the reference to the function we created (`add_func`).

To summarize what happened in this example, you saw an AWS AppSync function that implemented a request and response handler. The function was responsible for interacting with your data source. The request handler sent a `Scan` operation to AWS AppSync, instructing it on what operation to perform against your DynamoDB data source. The response handler returned the list of items (`ctx.result.items`). The list of items was then mapped to the `Post` GraphQL type automatically.

Creating basic mutation resolvers

This section will show you how to make a basic mutation resolver.

Console

1. Sign in to the AWS Management Console and open the AppSync console.
   a. In the **APIs dashboard**, choose your GraphQL API.
   b. In the **Sidebar**, choose **Schema**.
2. Under the **Resolvers** section and the **Mutation** type, choose **Attach** next to your field.

> **Note**
>
> In our example, we're attaching a resolver for `createPost`, which adds a `Post` object to our table. Let's assume we're using the same DynamoDB table from the last section. Its partition key is set to the `id` and is empty.

3. On the **Attach resolver** page, under **Resolver type**, choose `pipeline resolvers`. As a reminder, you can find more information about resolvers here. For **Resolver runtime**, choose APPSYNC_JS to enable the JavaScript runtime.
4. You can enable caching for this API. We recommend turning this feature off for now. Choose **Create**.
5. Choose **Add function**, then choose **Create new function**. Alternatively, you may see a **Create function** button to choose instead.

   a. Choose your data source. This should be the source whose data you will manipulate with the mutation.

   b. Enter a `Function name`.

   c. Under **Function code**, you'll need to implement the function's behavior. This is a mutation, so the request will ideally perform some state-changing operation on the invoked data source. The result will be processed by the response function.

   > **Note**
   >
   > `createPost` is adding, or "putting", a new `Post` in the table with our parameters as the data. We could add something like this:

```
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;
}
```

In this step, we also added `request` and `response` functions:

- `request`: The request handler accepts the context as the argument. The request handler return statement performs a `PutItem` command, which is a built-in DynamoDB operation (see here or here for examples). The `PutItem` command adds a `Post` object to our DynamoDB table by taking the partition `key` value (automatically generated by `util.autoid()`) and `attributes` from the context argument input (these are the values we will pass in our request). The `key` is the `id` and `attributes` are the `date` and `title` field arguments. They're both preformatted through the `util.dynamodb.toMapValues` helper to work with the DynamoDB table.
- `response`: The response accepts the updated context and returns the result of the request handler.

d. Choose **Create** after you're done.

6. Back on the resolver screen, under **Functions**, choose the **Add function** drop-down and add your function to your functions list.
7. Choose **Save** to update the resolver.

CLI

To add your function

- Create a function for your pipeline resolver using the `create-function` command.

  You'll need to enter a few parameters for this particular command:

  1. The `api-id` of your API.
  2. The `name` of the function in the AWS AppSync console.

3. The `data-source-name`, or the name of the data source the function will use. It must already be created and linked to your GraphQL API in the AWS AppSync service.
4. The `runtime`, or environment and language of the function. For JavaScript, the name must be `APPSYNC_JS`, and the runtime, `1.0.0`.
5. The `code`, or request and response handlers of your function. While you can type it in manually, it's far easier to add it to a .txt file (or a similar format) then pass it in as the argument.

> **Note**
>
> Our query code will be in a file passed in as the argument:
>
> ```
> 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;
> }
> ```

An example command may look like this:

```
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}
```

An output will be returned in the CLI. Here's an example:

```
{
    "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**
>
> Make sure you record the `functionId` somewhere as this will be used to attach the function to the resolver.

To create your resolver

- Create a pipeline function for `Mutation` by running the `create-resolver` command.

  You'll need to enter a few parameters for this particular command:

  1. The `api-id` of your API.
  2. The `type-name`, or the special object type in your schema (Query, Mutation, Subscription).
  3. The `field-name`, or the field operation inside the special object type you want to attach the resolver to.
  4. The `kind`, which specifies a unit or pipeline resolver. Set this to `PIPELINE` to enable pipeline functions.

5. The `pipeline-config`, or the function(s) to attach to the resolver. Make sure you know the `functionId` values of your functions. Order of listing matters.
6. The `runtime`, which was `APPSYNC_JS` (JavaScript). The `runtimeVersion` currently is `1.0.0`.
7. The `code`, which contains the before and after step.

> **Note**
>
> Our query code will be in a file passed in as the argument:

```
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;
}
```

An example command may look like this:

```
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}
```

An output will be returned in the CLI. Here's an example:

```
{
    "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**
>
> Before you use the CDK, we recommend reviewing the CDK's official documentation along with AWS AppSync's CDK reference.
> The steps listed below will only show a general example of the snippet used to add a particular resource. This is **not** meant to be a working solution in your production code. We also assume you already have a working app.

- To make a mutation, assuming you're in the same project, you can add it to the stack file like the query. Here's a modified function and resolver for a mutation that adds a new `Post` to the table:

```
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

Since this mutation and the query are similarly structured, we'll just explain the changes we made to make the mutation.

In the function, we changed the CFN id to `func-add-post` and name to `add_posts_func_1` to reflect the fact that we're adding `Posts` to the table. In the data source, we made a new association to our table (`add_ddb_table`) in the AWS AppSync console as `table-for-posts-2` because the `addDynamoDbDataSource` method requires it. Keep in mind, this new association is still using the same table we created earlier, but we now have two connections to it in the AWS AppSync console: one for the query as `table-for-posts` and one for the mutation as `table-for-posts-2`. The code was changed to add a `Post` by generating its `id` value automatically and accepting a client's input for the rest of the fields.
In the resolver, we changed the id value to `pipeline-resolver-create-posts` to reflect the fact that we're adding `Posts` to the table. To reflect the mutation in the schema, the type name was changed to `Mutation`, and the name, `createPost`. The pipeline config was set to our new mutation function `add_func_2`.

To summarize what's happening in this example, AWS AppSync automatically converts arguments defined in the `createPost` field from your GraphQL schema into DynamoDB operations. The example stores records in DynamoDB using a key of `id`, which is automatically created using our `util.autoId()` helper. All of the other fields you pass to the context arguments (`ctx.args.input`) from requests made in the AWS AppSync console or otherwise will be stored as the table's attributes. Both the key and the attributes are automatically mapped to a compatible DynamoDB format using the `util.dynamodb.toMapValues(values)` helper.

AWS AppSync also supports test and debug workflows for editing resolvers. You can use a mock `context` object to see the transformed value of the template before invoking it. Optionally, you can view the full request to a data source interactively when you run a query. For more information, see Test and debug resolvers (JavaScript) and Monitoring and logging.

Advanced resolvers

If you are following the optional pagination section in Designing your schema, you still need to add your resolver to your request to make use of pagination. Our example used a query pagination called `getPosts` to return only a portion of the things requested at a time. Our resolver's code on that field may look like this:

```
/**
 * 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 };
}
```

In the request, we pass in the context of the request. Our `limit` is *20*, meaning we return up to 20 `Posts` in the first query. Our `nextToken` cursor is fixed to the first `Post` entry in the data source. These are passed to the args. The request then performs a scan from the first `Post` up to the scan limit number. The data source stores the result in the context, which is passed to the response. The response returns the `Posts` it retrieved, then sets the `nextToken` is set to the `Post` entry right after the limit. The next request is sent out to do the exact same thing but starting at the offset right after the first query. Keep in mind that these sorts of requests are done sequentially and not in parallel.

Testing and debugging resolvers in AWS AppSync (JavaScript)

AWS AppSync executes resolvers on a GraphQL field against a data source. When working with pipeline resolvers, functions interact with your data sources. As described in the JavaScript resolvers overview, functions communicate with data sources by using request and response handlers written in JavaScript and running on the `APPSYNC_JS` runtime. This enables you to provide custom logic and conditions before and after communicating with the data source.

To help developers write, test, and debug these resolvers, the AWS AppSync console also provides tools to create a GraphQL request and response with mock data down to the individual field resolver. Additionally, you can perform queries, mutations, and subscriptions in the AWS AppSync console and see a detailed log stream of the entire request from Amazon CloudWatch. This includes results from the data source.

Testing with mock data

When a GraphQL resolver is invoked, it contains a `context` object that has relevant information about the request. This includes arguments from a client, identity information, and data from

the parent GraphQL field. It also stores the results from the data source, which can be used in the response handler. For more information about this structure and the available helper utilities to use when programming, see the Resolver context object reference.

When writing or editing a resolver function, you can pass a *mock* or *test context* object into the console editor. This enables you to see how both the request and the response handlers evaluate without actually running against a data source. For example, you can pass a test `firstname: Shaggy` argument and see how it evaluates when using `ctx.args.firstname` in your template code. You could also test the evaluation of any utility helpers such as `util.autoId()` or `util.time.nowISO8601()`.

Testing resolvers

This example will use the AWS AppSync console to test resolvers.

1. Sign in to the AWS Management Console and open the AppSync console.
   a. In the **APIs dashboard**, choose your GraphQL API.
   b. In the **Sidebar**, choose **Functions**.
2. Choose an existing function.
3. At the top of the **Update function** page, choose **Select test context**, then choose **Create new context**.
4. Select a sample context object or populate the JSON manually in the **Configure test context** window below.
5. Enter a **Text context name**.
6. Choose the **Save** button.
7. To evaluate your resolver using this mocked context object, choose **Run Test**.

For a more practical example, suppose you have an app storing a GraphQL type of Dog that uses automatic ID generation for objects and stores them in Amazon DynamoDB. You also want to write some values from the arguments of a GraphQL mutation and allow only specific users to see a response. The following snippet shows what the schema might look like:

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

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

You can write an AWS AppSync function and add it to your addDog resolver to handle the mutation. To test your AWS AppSync function, you can populate a context object like the following example. The following has arguments from the client of `name` and `age`, and a `username` populated in the `identity` object:

```
{
    "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"
    }
}
```

You can test your AWS AppSync function using the following code:

```
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();
}
```

The evaluated request and response handler has the data from your test context object and the generated value from `util.autoId()`. Additionally, if you were to change the `username` to a value other than `Nadia`, the results won't be returned because the authorization check would fail. For more information about fine-grained access control, see Authorization use cases.

Testing request and response handlers with AWS AppSync's APIs

You can use the `EvaluateCode` API command to remotely test your code with mocked data. To get started with the command, make sure you have added the `appsync:evaluateMappingCode` permission to your policy. For example:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateCode",
            "Resource": "arn:aws:appsync:us-east-1:111122223333:*"
        }
    ]
}
```

You can leverage the command by using the AWS CLI or AWS SDKs. For example, take the `Dog` schema and its AWS AppSync function request and response handlers from the previous section. Using the CLI on your local station, save the code to a file named `code.js`, then save the `context` object to a file named `context.json`. From your shell, run the following command:

```
$ aws appsync evaluate-code \
  --code file://code.js \
  --function response \
```

```
  --context file://context.json \
  --runtime name=APPSYNC_JS,runtimeVersion=1.0.0
```

The response contains an `evaluationResult` containing the payload returned by your handler. It also contains a `logs` object, that holds the list of logs that were generated by your handler during the evaluation. This makes it easy to debug your code execution and see information about your evaluation to help troubleshoot. For example:

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

The `evaluationResult` can be parsed as JSON, which gives:

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

Using the SDK, you can easily incorporate tests from your favorite test suite to validate your handlers' behavior. We recommend creating tests using the Jest Testing Framework, but any testing suite works. The following snippet shows a hypothetical validation run. Note that we expect the evaluation response to be valid JSON, so we use `JSON.parse` to retrieve JSON from the string response:

```
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)
})
```

This yields the following result:

```
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
```

Debugging a live query

There's no substitute for an end-to-end test and logging to debug a production application. AWS AppSync lets you log errors and full request details using Amazon CloudWatch. Additionally, you can use the AWS AppSync console to test GraphQL queries, mutations, and subscriptions and live stream log data for each request back into the query editor to debug in real time. For subscriptions, the logs display connection-time information.

To perform this, you need to have Amazon CloudWatch logs enabled in advance, as described in Monitoring and logging. Next, in the AWS AppSync console, choose the **Queries** tab and then enter a valid GraphQL query. In the lower-right section, click and drag the **Logs** window to open the logs view. At the top of the page, choose the play arrow icon to run your GraphQL query. In a few moments, your full request and response logs for the operation are streamed to this section and you can view them in the console.

Configuring and using pipeline resolvers in AWS AppSync (JavaScript)

AWS AppSync executes resolvers on a GraphQL field. In some cases, applications require executing multiple operations to resolve a single GraphQL field. With pipeline resolvers, developers can now compose operations called Functions and execute them in sequence. Pipeline resolvers are useful for applications that, for instance, require performing an authorization check before fetching data for a field.

For more information about the architecture of a JavaScript pipeline resolver, see the JavaScript resolvers overview.

Step 1: Creating a pipeline resolver

In the AWS AppSync console, go to the **Schema** page.

Save the following schema:

```
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
}
```

We are going to wire a pipeline resolver to the **signUp** field on the **Mutation** type. In the **Mutation** type on the right side, choose **Attach** next to the `signUp` mutation field. Set the resolver to `pipeline resolver` and the `APPSYNC_JS` runtime, then create the resolver.

Our pipeline resolver signs up a user by first validating the email address input and then saving the user in the system. We are going to encapsulate the email validation inside a **validateEmail** function and the saving of the user inside a **saveUser** function. The **validateEmail** function executes first, and if the email is valid, then the **saveUser** function executes.

The execution flow will be as follows:

1. Mutation.signUp resolver request handler

2. validateEmail function
3. saveUser function
4. Mutation.signUp resolver response handler

Because we will probably reuse the **validateEmail** function in other resolvers on our API, we want to avoid accessing `ctx.args` because these will change from one GraphQL field to another. Instead, we can use the `ctx.stash` to store the email attribute from the `signUp(input: Signup)` input field argument.

Update your resolver code by replacing your request and response functions:

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

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

Choose **Create** or **Save** to update the resolver.

Step 2: Creating a function

From the pipeline resolver page, in the **Functions** section, click on **Add function**, then **Create new function**. It is also possible to create functions without going through the resolver page; to do this, in the AWS AppSync console, go to the **Functions** page. Choose the **Create function** button. Let's create a function that checks if an email is valid and comes from a specific domain. If the email is not valid, the function raises an error. Otherwise, it forwards whatever input it was given.

Make sure you have created a data source of the **NONE** type. Choose this data source in the **Data source name** list. For the **function name**, enter in `validateEmail`. In the **function code** area, overwrite everything with this snippet:

```
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;
}
```

Review your inputs, then choose **Create**. We just created our **validateEmail** function. Repeat these steps to create the **saveUser** function with the following code (For the sake of simplicity, we use a **NONE** data source and pretend the user has been saved in the system after the function executes.):

```
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;
}
```

We just created our **saveUser** function.

Step 3: Adding a function to a pipeline resolver

Our functions should have been added automatically to the pipeline resolver we just created. If this wasn't the case, or you created the functions through the **Functions** page, you can click on **Add function** back on the `signUp` resolver page to attach them. Add both the **validateEmail** and **saveUser** functions to the resolver. The **validateEmail** function should be placed before the **saveUser** function. As you add more functions, you can use the **move up** and **move down** options to reorganize the order of execution of your functions. Review your changes, then choose **Save**.

Step 4: Running a query

In the AWS AppSync console, go to the **Queries** page. In the explorer, ensure that you're using your mutation. If you aren't, choose `Mutation` in the drop-down list, then choose +. Enter the following query:

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

This should return something like:

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

We have successfully signed up our user and validated the input email using a pipeline resolver.

Creating basic queries (VTL)

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

GraphQL resolvers connect the fields in a type's schema to a data source. Resolvers are the mechanism by which requests are fulfilled. AWS AppSync can automatically create and connect resolvers from a schema or create a schema and connect resolvers from an existing table without you needing to write any code.

Resolvers in AWS AppSync use JavaScript to convert a GraphQL expression into a format the data source can use. Alternatively, mapping templates can be written in Apache Velocity Template Language (VTL) to convert a GraphQL expression into a format the data source can use.

This section will show you how to configure resolvers using VTL. An introductory tutorial-style programming guide for writing resolvers can be found in Resolver mapping template programming guide, and helper utilities available to use when programming can be found in Resolver mapping template context reference. AWS AppSync also has built-in test and debug flows that you can use when you're editing or authoring from scratch. For more information, see Test and debug resolvers.

We recommend following this guide before attempting to to use any of the aforementioned tutorials.

In this section, we will walk through how to create a resolver, add a resolver for mutations, and use advanced configurations.

Create your first resolver

Following the examples from the previous sections, the first step is to create a resolver for your `Query` type.

Console

1. Sign in to the AWS Management Console and open the AppSync console.
   a. In the **APIs dashboard**, choose your GraphQL API.
   b. In the **Sidebar**, choose **Schema**.
2. On the right-hand side of the page, there's a window called **Resolvers**. This box contains a list of the types and fields as defined in your **Schema** window on the left-hand side of the page. You're able to attach resolvers to fields. For example, under the **Query** type, choose **Attach** next to the `getTodos` field.
3. On the **Create Resolver** page, choose the data source you created in the Attaching a data source guide. In the **Configure mapping templates** window, you can choose both the generic request and response mapping templates using the drop-down list to the right or write your own.

> **Note**
>
> The pairing of a request mapping template to a response mapping template is called a unit resolver. Unit resolvers are typically meant to perform rote operations;

we recommend using them only for singular operations with a small number of data sources. For more complex operations, we recommend using pipeline resolvers, which can execute multiple operations with multiple data sources sequentially.
For more information about the difference between request and response mapping templates, see Unit resolvers.
For more information about using pipeline resolvers, see Pipeline resolvers.

4. For common use cases, the AWS AppSync console has built-in templates that you can use for getting items from data sources (e.g., all item queries, individual lookups, etc.). For example, on the simple version of the schema from Designing your schema where `getTodos` didn't have pagination, the request mapping template for listing items is as follows:

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

5. You always need a response mapping template to accompany the request. The console provides a default with the following passthrough value for lists:

```
$util.toJson($ctx.result.items)
```

In this example, the `context` object (aliased as `$ctx`) for lists of items has the form `$context.result.items`. If your GraphQL operation returns a single item, it would be `$context.result`. AWS AppSync provides helper functions for common operations, such as the `$util.toJson` function listed previously, to format responses properly. For a full list of functions, see Resolver mapping template utility reference.

6. Choose **Save Resolver**.

API

1. Create a resolver object by calling the `CreateResolver` API.
2. You can modify your resolver's fields by calling the `UpdateResolver` API.

CLI

1. Create a resolver by running the `create-resolver` command.

   You'll need to type in 6 parameters for this particular command:

   1. The `api-id` of your API.
   2. The `type-name` of the type that you want to modify in your schema. In the console example, this was `Query`.
   3. The `field-name` of the field that you want to modify in your type. In the console example, this was `getTodos`.
   4. The `data-source-name` of the data source you created in the Attaching a data source guide.
   5. The `request-mapping-template`, which is the body of the request. In the console example, this was:

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

   6. The `response-mapping-template`, which is the body of the response. In the console example, this was:

      ```
      $util.toJson($ctx.result.items)
      ```

   An example command may look like this:

   ```
   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)"
   ```

   An output will be returned in the CLI. Here's an example:

   ```
   {
       "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. To modify a resolver's fields and/or mapping templates, run the `update-resolver` command.

   With the exception of the `api-id` parameter, the parameters used in the `create-resolver` command will be overwritten by the new values from the `update-resolver` command.

Adding a resolver for mutations

The next step is to create a resolver for your `Mutation` type.

Console

1. Sign in to the AWS Management Console and open the AppSync console.
   a. In the **APIs dashboard**, choose your GraphQL API.
   b. In the **Sidebar**, choose **Schema**.
2. Under the **Mutation** type, choose **Attach** next to the addTodo field.
3. On the **Create Resolver** page, choose the data source you created in the Attaching a data source guide.
4. In the **Configure mapping templates** window, you'll need to modify the request template because this is a mutation where you're adding a new item to DynamoDB. Use the following request mapping template:

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
```

```
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

5. AWS AppSync automatically converts arguments defined in the addTodo field from your GraphQL schema into DynamoDB operations. The previous example stores records in DynamoDB using a key of `id`, which is passed through from the mutation argument as `$ctx.args.id`. All of the other fields you pass through are automatically mapped to DynamoDB attributes with `$util.dynamodb.toMapValuesJson($ctx.args)`.

   For this resolver, use the following response mapping template:

   ```
   $util.toJson($ctx.result)
   ```

   AWS AppSync also supports test and debug workflows for editing resolvers. You can use a mock `context` object to see the transformed value of the template before invoking. Optionally, you can view the full request execution to a data source interactively when you run a query. For more information, see Test and debug resolvers and Monitoring and logging.
6. Choose **Save Resolver**.

API

You can also do this with APIs by utilizing the commands in the Create your first resolver section and the parameter details from this section.

CLI

You can also do this in the CLI by utilizing the commands in the Create your first resolver section and the parameter details from this section.

At this point, if you're not using the advanced resolvers you can begin using your GraphQL API as outlined in Using your API.

Advanced resolvers

If you are following the Advanced section and you're building a sample schema in Designing your schema to do a paginated scan, use the following request template for the `getTodos` field instead:

```
{
```

```
    "version" : "2017-02-28",
    "operation" : "Scan",
    "limit": $util.defaultIfNull(${ctx.args.limit}, 20),
    "nextToken": $util.toJson($util.defaultIfNullOrBlank($ctx.args.nextToken, null))
}
```

For this pagination use case, the response mapping is more than just a passthrough because it must contain both the *cursor* (so that the client knows what page to start at next) and the result set. The mapping template is as follows:

```
{
    "todos": $util.toJson($context.result.items),
    "nextToken": $util.toJson($context.result.nextToken)
}
```

The fields in the preceding response mapping template should match the fields defined in your `TodoConnection` type.

For the case of relations where you have a `Comments` table and you're resolving the comments field on the `Todo` type (which returns a type of `[Comment]`), you can use a mapping template that runs a query against the second table. To do this, you must have already created a data source for the `Comments` table as outlined in Attaching a data source.

> **Note**
>
> We're using a query operation against a second table for illustrative purposes only. You could use another operation against DynamoDB instead. In addition, you could pull the data from another data source, such as AWS Lambda or Amazon OpenSearch Service, because the relation is controlled by your GraphQL schema.

Console

1. Sign in to the AWS Management Console and open the AppSync console.
   a. In the **APIs dashboard**, choose your GraphQL API.
   b. In the **Sidebar**, choose **Schema**.
2. Under the **Todo** type, choose **Attach** next to the `comments` field.

3. On the **Create Resolver** page, choose your **Comments** table data source. The default name for the **Comments** table from the quickstart guides is `AppSyncCommentTable`, but it may vary depending on what name you gave it.
4. Add the following snippet to your request mapping template:

```
{
    "version": "2017-02-28",
    "operation": "Query",
    "index": "todoid-index",
    "query": {
        "expression": "todoid = :todoid",
        "expressionValues": {
            ":todoid": {
                "S": $util.toJson($context.source.id)
            }
        }
    }
}
```

5. The `context.source` references the parent object of the current field that's being resolved. In this example, `source.id` refers to the individual `Todo` object, which is then used for the query expression.

   You can use the passthrough response mapping template as follows:

```
$util.toJson($ctx.result.items)
```

6. Choose **Save Resolver**.
7. Finally, back on the **Schema** page in the console, attach a resolver to the `addComment` field, and specify the data source for the `Comments` table. The request mapping template in this case is a simple `PutItem` with the specific `todoid` that is commented on as an argument, but you use the `$utils.autoId()` utility to create a unique sort key for the comment as follows:

```
{
    "version": "2017-02-28",
    "operation": "PutItem",
    "key": {
        "todoid": { "S": $util.toJson($context.arguments.todoid) },
        "commentid": { "S": "$util.autoId()" }
```

```
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

Use a passthrough response template as follows:

```
$util.toJson($ctx.result)
```

API

You can also do this with APIs by utilizing the commands in the Create your first resolver section and the parameter details from this section.

CLI

You can also do this in the CLI by utilizing the commands in the Create your first resolver section and the parameter details from this section.

Disabling VTL mapping templates with direct Lambda resolvers (VTL)

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

With direct Lambda resolvers, you can circumvent the use of VTL mapping templates when using AWS Lambda data sources. AWS AppSync can provide a default payload to your Lambda function as well as a default translation from a Lambda function's response to a GraphQL type. You can choose to provide a request template, a response template, or neither and AWS AppSync will handle it accordingly.

To learn more about the default request payload and response translation that AWS AppSync provides, see the Direct Lambda resolver reference. For more information on setting up an AWS Lambda data source and setting up an IAM Trust Policy, see Attaching a data source.

Configure direct Lambda resolvers

The following sections will show you how to attach Lambda data sources and add Lambda resolvers to your fields.

Add a Lambda data source

Before you can activate direct Lambda resolvers, you must add a Lambda data source.

Console

1. Sign in to the AWS Management Console and open the AppSync console.
   a. In the **APIs dashboard**, choose your GraphQL API.
   b. In the **Sidebar**, choose **Data sources**.
2. Choose **Create data source**.
   a. For **Data source name**, enter a name for your data source, such as **`myFunction`**.
   b. For **Data source type**, choose **AWS Lambda function**.
   c. For **Region**, choose the appropriate region.
   d. For **Function ARN**, choose the Lambda function from the dropdown list. You can search for the function name or manually enter the ARN of the function you want to use.
   e. Create a new IAM role (recommended) or choose an existing role that has the `lambda:invokeFunction` IAM permission. Existing roles need a trust policy, as explained in the Attaching a data source section.

      The following is an example IAM policy that has the required permissions to perform operations on the resource:

      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. Choose the **Create** button.

CLI

1. Create a data source object by running the `create-data-source` command.

   You'll need to type in 4 parameters for this particular command:

   1. The `api-id` of your API.
   2. The `name` of your data source. In the console example, this is the **Data source name**.
   3. The `type` of data source. In the console example, this is **AWS Lambda function**.
   4. The `lambda-config`, which is the **Function ARN** in the console example.

   > **Note**
   >
   > There are other parameters such as `Region` that must be configured but will usually default to your CLI configuration values.

   An example command may look like this:

```
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
```

   An output will be returned in the CLI. Here's an example:

```
{
    "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. To modify a data source's attributes, run the `update-data-source` command.

   With the exception of the `api-id` parameter, the parameters used in the `create-data-source` command will be overwritten by the new values from the `update-data-source` command.

Activate direct Lambda resolvers

After creating a Lambda data source and setting up the appropriate IAM role to allow AWS AppSync to invoke the function, you can link it to a resolver or pipeline function.

Console

1. Sign in to the AWS Management Console and open the AppSync console.
   a. In the **APIs dashboard**, choose your GraphQL API.
   b. In the **Sidebar**, choose **Schema**.
2. In the **Resolvers** window, choose a field or operation and then select the **Attach** button.
3. In the **Create new resolver** page, choose the Lambda function from the dropdown list.
4. In order to leverage direct Lambda resolvers, confirm that request and response mapping templates are disabled in the **Configure mapping templates** section.
5. Choose the **Save Resolver** button.

CLI

- Create a resolver by running the `create-resolver` command.

  You'll need to type in 6 parameters for this particular command:

  1. The `api-id` of your API.
  2. The `type-name` of the type in your schema.
  3. The `field-name` of the field in your schema.

4. The `data-source-name`, or your Lambda function's name.
5. The `request-mapping-template`, which is the body of the request. In the console example, this was disabled:

```
" "
```

6. The `response-mapping-template`, which is the body of the response. In the console example, this was also disabled:

```
" "
```

An example command may look like this:

```
aws appsync create-resolver --api-id abcdefghijklmnopqrstuvwxyz --type-name
 Subscription --field-name onCreateTodo --data-source-name LambdaTest --request-
mapping-template " " --response-mapping-template " "
```

An output will be returned in the CLI. Here's an example:

```
{
    "resolver": {
        "resolverArn": "arn:aws:appsync:us-west-2:102847592837:apis/
abcdefghijklmnopqrstuvwxyz/types/Subscription/resolvers/onCreateTodo",
        "typeName": "Subscription",
        "kind": "UNIT",
        "fieldName": "onCreateTodo",
        "dataSourceName": "LambdaTest"
    }
}
```

When you disable your mapping templates, there are several additional behaviors that will occur in AWS AppSync:

- By disabling a mapping template, you are signalling to AWS AppSync that you accept the default data translations specified in the Direct Lambda resolver reference.
- By disabling the request mapping template, your Lambda data source will receive a payload consisting of the entire Context object.

- By disabling the response mapping template, the result of your Lambda invocation will be translated depending on the version of the request mapping template or if the request mapping template is also disabled.

Testing and debugging resolvers in AWS AppSync (VTL)

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

AWS AppSync executes resolvers on a GraphQL field against a data source. As described in Resolver mapping template overview, resolvers communicate with data sources by using a templating language. This enables you to customize the behavior and apply logic and conditions before and after communicating with the data source. For an introductory tutorial-style programming guide for writing resolvers, see the Resolver mapping template programming guide.

To help developers write, test, and debug these resolvers, the AWS AppSync console also provides tools to create a GraphQL request and response with mock data down to the individual field resolver. Additionally, you can perform queries, mutations, and subscriptions in the AWS AppSync console and see a detailed log stream from Amazon CloudWatch of the entire request. This includes results from a data source.

Testing with mock data

When a GraphQL resolver is invoked, it contains a `context` object that contains information about the request. This includes arguments from a client, identity information, and data from the parent GraphQL field. It also contains the results from the data source, which can be used in the response template. For more information about this structure and the available helper utilities to use when programming, see the Resolver Mapping Template Context Reference.

When writing or editing a resolver, you can pass a *mock* or *test context* object into the console editor. This enables you to see how both the request and the response templates evaluate without actually running against a data source. For example, you can pass a test `firstname: Shaggy` argument and see how it evaluates when using `$ctx.args.firstname` in your template code. You could also test the evaluation of any utility helpers such as `$util.autoId()` or `util.time.nowISO8601()`.

Testing resolvers

This example will use the AWS AppSync console to test resolvers.

1. Sign in to the AWS Management Console and open the AppSync console.
   a. In the **APIs dashboard**, choose your GraphQL API.
   b. In the **Sidebar**, choose **Schema**.
2. If you haven't done so already, under the type and next to the field, choose **Attach** to add your resolver.

   For more information on how to build a conplete resolver, see Configuring resolvers.

   Otherwise, select the resolver that's already in the field.
3. At the top of the **Edit resolver** page, choose **Select test context**, choose **Create new context**.
4. Select a sample context object or populate the JSON manually in the **Execution context** window below.
5. Enter in a **Text context name**.
6. Choose the **Save** button.
7. At the top of the **Edit Resolver** page, choose **Run test**.

For a more practical example, suppose you have an app storing a GraphQL type of Dog that uses automatic ID generation for objects and stores them in Amazon DynamoDB. You also want to write some values from the arguments of a GraphQL mutation, and allow only specific users to see a response. The following shows what the schema might look like:

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

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

When you add a resolver for the addDog mutation, you can populate a context object like the following example. The following has arguments from the client of name and age, and a username populated in the identity object:

```
{
    "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"
    }
}
```

You can test this using the following request and response mapping templates:

Request Template

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : { "S" : "$util.autoId()" }
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

Response Template

```
#if ($context.identity.username == "Nadia")
  $util.toJson($ctx.result)
#else
  $util.unauthorized()
#end
```

The evaluated template has the data from your test context object and the generated value from `$util.autoId()`. Additionally, if you were to change the `username` to a value other than `Nadia`, the results won't be returned because the authorization check would fail. For more information about fine grained access control, see Authorization use cases.

Testing mapping templates with AWS AppSync's APIs

You can use the `EvaluateMappingTemplate` API command to remotely test your mapping templates with mocked data. To get started with the command, make sure you have added the `appsync:evaluateMappingTemplate` permission to your policy. For example:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateMappingTemplate",
            "Resource": "arn:aws:appsync:us-east-1:111122223333:*"
        }
    ]
}
```

You can leverage the command by using the AWS CLI or AWS SDKs. For example, take the Dog schema and its request/response mapping templates from the previous section. Using the CLI on your local station, save the request template to a file named `request.vtl`, then save the `context` object to a file named `context.json`. From your shell, run the following command:

```
aws appsync evaluate-mapping-template --template file://request.vtl --context file://
context.json
```

The command returns the following response:

```
{
  "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"
```

```
}
```

The `evaluationResult` contains the results of testing your provided template with the provided `context`. You can also test your templates using the AWS SDKs. Here's an example using the AWS SDK for JavaScript V2:

```
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))
```

Using the SDK, you can easily incorporate tests from your favorite test suite to validate your template's behavior. We recommend creating tests using the Jest Testing Framework, but any testing suite works. The following snippet shows a hypothetical validation run. Note that we expect the evaluation response to be valid JSON, so we use `JSON.parse` to retrieve JSON from the string response:

```
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)
})
```

This yields the following result:

```
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
```

Debugging a live query

There's no substitute for an end-to-end test and logging to debug a production application. AWS AppSync lets you log errors and full request details using Amazon CloudWatch. Additionally, you can use the AWS AppSync console to test GraphQL queries, mutations, and subscriptions and live stream log data for each request back into the query editor to debug in real time. For subscriptions, the logs display connection-time information.

To perform this, you need to have Amazon CloudWatch logs enabled in advance, as described in Monitoring and logging. Next, in the AWS AppSync console, choose the **Queries** tab and then enter a valid GraphQL query. In the lower-right section, click and drag the **Logs** window to open the logs view. At the top of the page, choose the play arrow icon to run your GraphQL query. In a few moments, your full request and response logs for the operation are streamed to this section and you can view then in the console.

Configuring and using pipeline resolvers in AWS AppSync (VTL)

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

AWS AppSync executes resolvers on a GraphQL field. In some cases, applications require executing multiple operations to resolve a single GraphQL field. With pipeline resolvers, developers can now compose operations called Functions and execute them in sequence. Pipeline resolvers are useful for applications that, for instance, require performing an authorization check before fetching data for a field.

A pipeline resolver is composed of a **Before** mapping template, an **After** mapping template, and a list of Functions. Each Function has a **request** and **response** mapping template that it executes against a data source. As a pipeline resolver delegates execution to a list of functions, it is therefore not linked to any data source. Unit resolvers and functions are primitives that execute operations against data sources. See the Resolver mapping template overview for more information.

Step 1: Creating a pipeline resolver

In the AWS AppSync console, go to the **Schema** page.

Save the following schema:

```
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
}
```

We are going to wire a pipeline resolver to the **signUp** field on the **Mutation** type. In the **Mutation** type on the right side, choose **Attach** next to the `signUp` mutation field. On the create resolver page, click on **Actions**, then **Update runtime**. Choose `Pipeline Resolver`, then choose VTL, then choose **Update**. The page should now show three sections: a **Before mapping template** text area, a **Functions** section, and an **After mapping template** text area.

Our pipeline resolver signs up a user by first validating the email address input and then saving the user in the system. We are going to encapsulate the email validation inside a **validateEmail** function, and the saving of the user inside a **saveUser** function. The **validateEmail** function executes first, and if the email is valid, then the **saveUser** function executes.

The execution flow will be as follow:

1. Mutation.signUp resolver request mapping template
2. validateEmail function
3. saveUser function
4. Mutation.signUp resolver response mapping template

Because we will probably reuse the **validateEmail** function in other resolvers on our API, we want to avoid accessing `$ctx.args` because these will change from one GraphQL field to another. Instead, we can use the `$ctx.stash` to store the email attribute from the `signUp(input: Signup)` input field argument.

BEFORE mapping template:

```
## store email input field into a generic email key
$util.qr($ctx.stash.put("email", $ctx.args.input.email))
{}
```

The console provides a default passthrough **AFTER** mapping template that will we use:

```
$util.toJson($ctx.result)
```

Choose **Create** or **Save** to update the resolver.

Step 2: Creating a function

From the pipeline resolver page, in the **Functions** section, click on **Add function**, then **Create new function**. It is also possible to create functions without going through the resolver page; to do this, in the AWS AppSync console, go to the **Functions** page. Choose the **Create function** button. Let's create a function that checks if an email is valid and comes from a specific domain. If the email is not valid, the function raises an error. Otherwise, it forwards whatever input it was given.

On the new function page, choose **Actions**, then **Update runtime**. Choose VTL, then **Update**. Make sure you have created a data source of the **NONE** type. Choose this data source in the **Data source**

name list. For **function name**, enter in `validateEmail`. In the **function code** area, overwrite everything with this snippet:

```
#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}) }
}
```

Paste this into the response mapping template:

```
$util.toJson($ctx.result)
```

Review your changes, then choose **Create**. We just created our **validateEmail** function. Repeat these steps to create the **saveUser** function with the following request and response mapping templates (For the sake of simplicity, we use a **NONE** data source and pretend the user has been saved in the system after the function executes.):

Request mapping template:

```
## $ctx.prev.result contains the signup input values. We could have also
## used $ctx.args.input.
{
    "payload": $util.toJson($ctx.prev.result)
}
```

Response mapping template:

```
## 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)
```

We just created our **saveUser** function.

Step 3: Adding a function to a pipeline resolver

Our functions should have been added automatically to the pipeline resolver we just created. If this wasn't the case, or you created the functions through the **Functions** page, you can click on **Add**

function on the resolver page to attach them. Add both the **validateEmail** and **saveUser** functions to the resolver. The **validateEmail** function should be placed before the **saveUser** function. As you add more functions, you can use the **move up** and **move down** options to reorganize the order of execution of your functions. Review your changes, then choose **Save**.

Step 4: Executing a query

In the AWS AppSync console, go to the **Queries** page. In the explorer, ensure that you're using your mutation. If you aren't, choose `Mutation` in the drop-down list, then choose +. Enter the following query:

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

This should return something like:

```
{
  "data": {
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "email": "nadia@myvaliddomain.com"
    }
  }
}
```

We have successfully signed up our user and validated the input email using a pipeline resolver. To follow a more complete tutorial focusing on pipeline resolvers, you can go to Tutorial: Pipeline Resolvers

Using an AWS AppSync API with the AWS CDK

> **Tip**
>
> Before you use the CDK, we recommend reviewing the CDK's official documentation along with AWS AppSync's CDK reference.
> We also recommend ensuring that your AWS CLI and NPM installations are working on your system.

In this section, we're going to create a simple CDK application that can add and fetch items from a DynamoDB table. This is meant to be a quickstart example using some of the code from the Designing your schema, Attaching a data source, and Configuring resolvers (JavaScript) sections.

Setting up a CDK project

> **Warning**
>
> These steps may not be completely accurate depending on your environment. We're assuming your system has the necessary utilities installed, a way to interface with AWS services, and proper configurations in place.

The first step is installing the AWS CDK. In your CLI, you can enter the following command:

```
npm install -g aws-cdk
```

Next, you need to create a project directory, then navigate to it. An example set of commands to create and navigate to a directory is:

```
mkdir example-cdk-app
cd example-cdk-app
```

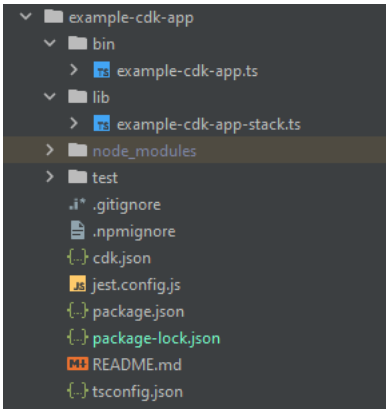
Next, you need to create an app. Our service primarily uses TypeScript. In your project directory, enter the following command:

```
cdk init app --language typescript
```

When you do this, a CDK app along with its initialization files will be installed:

```
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!
```

Your project structure may look like this:



You'll notice we have several important directories:

- `bin`: The initial bin file will create the app. We won't touch this in this guide.
- `lib`: The lib directory contains your stack files. You can think of stack files as individual units of execution. Constructs will be inside our stack files. Basically, these are resources for a service that will be spun up in AWS CloudFormation when the app is deployed. This is where most of our coding will happen.
- `node_modules`: This directory is created by NPM and contains all package dependencies you installed using the `npm` command.

Our initial stack file may contain something like this:

```
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)
    // });
  }
}
```

This is the boilerplate code to create a stack in our app. Most of our code in this example will go inside the scope of this class.

To verify that your stack file is in the app, in your app's directory, run the following command in the terminal:

```
cdk ls
```

A list of your stacks should appear. If it doesn't, then you may need to run through the steps again or check the official documentation for help.

If you want to build your code changes before deploying, you can always run the following command in the terminal:

```
npm run build
```

And, to see the changes before deploying:

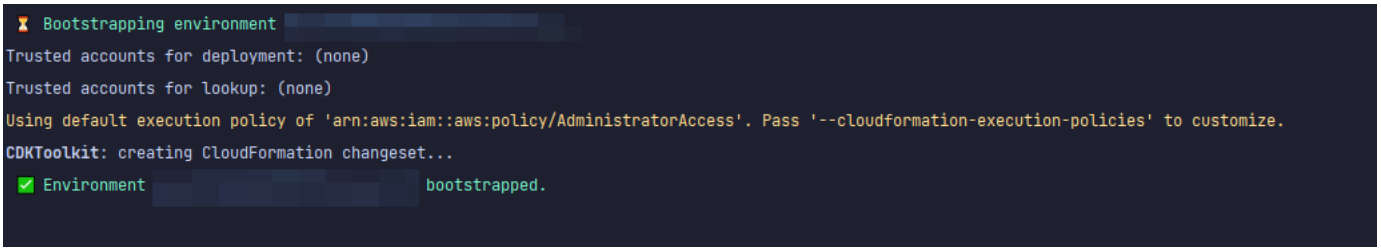
```
cdk diff
```

Before we add our code to the stack file, we're going to perform a bootstrap. Bootstrapping allows us to provision resources for the CDK before the app deploys. More information about this process can be found here. To create a bootstrap, the command is:

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

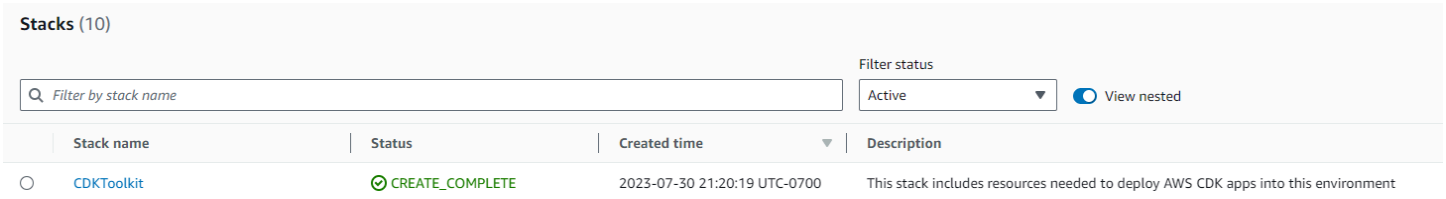
> **Tip**
>
> This step requires several IAM permissions in your account. Your bootstrap will be denied if you don't have them. If this happens, you may have to delete incomplete resources caused by the bootstrap such as the S3 bucket it generates.

Bootstrap will spin up several resources. The final message will look like this:

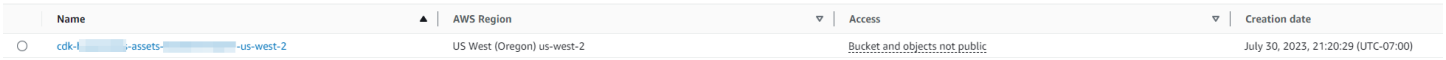


This is done once per account per Region, so you won't have to do this often. The main resources of the bootstrap are the AWS CloudFormation stack and the Amazon S3 bucket.

The Amazon S3 bucket is used to store files and IAM roles that grant permissions needed to perform deployments. The required resources are defined in an AWS CloudFormation stack, called the bootstrap stack, which is usually named `CDKToolkit`. Like any AWS CloudFormation stack, it appears in the AWS CloudFormation console once it has been deployed:



The same can be said for the bucket:



To import the services we need in our stack file, we can use the following command:

```
npm install aws-cdk-lib # V2 command
```

> **Tip**
>
> If you're having trouble with V2, you could install the individual libraries using V1 commands:

```
npm install @aws-cdk/aws-appsync @aws-cdk/aws-dynamodb
```

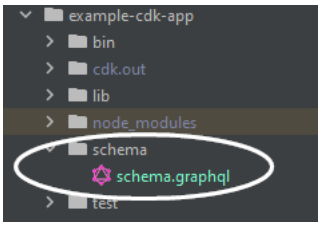
We don't recommend this because V1 has been deprecated.

Implementing a CDK project - Schema

We can now start implementing our code. First, we must create our schema. You can simply create a `.graphql` file in your app:

```
mkdir schema
touch schema.graphql
```

In our example, we included a top-level directory called `schema` containing our `schema.graphql`:



Inside our schema, let's include a simple example:

```
input CreatePostInput {
    title: String
    content: String
}

type Post {
    id: ID!
    title: String
    content: String
}

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

type Query {
    getPost: [Post]
```

```
}
```

Back in our stack file, we need to make sure the following import directives are defined:

```
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';
```

Inside the class, we'll add code to make our GraphQL API and connect it to our `schema.graphql` file:

```
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'),
    });
  }
}
```

We'll also add some code to print out the GraphQL URL, API key, and Region:

```
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
    });
  }
}
```

At this point, we'll use deploy our app again:

```
cdk deploy
```

This is the result:

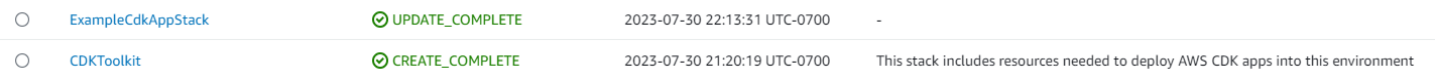
It appears our example was successful, but let's check the AWS AppSync console just to confirm:

It appears our API was created. Now, we'll check the schema attached to the API:

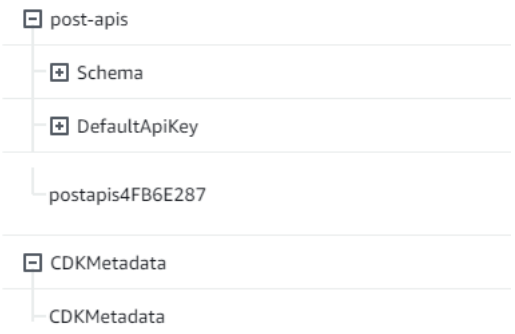
Schema

```
input CreatePostInput {
  title: String
  date: AWSDateTime
}

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

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

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

This appears to match up with our schema code, so it was successful. Another way to confirm this from a metadata viewpoint is to look at the AWS CloudFormation stack:



When we deploy our CDK app, it goes through AWS CloudFormation to spin up resources like the bootstrap. Each stack within our app maps 1:1 with an AWS CloudFormation stack. If you go back to the stack code, the stack name was grabbed from the class name `ExampleCdkAppStack`. You can see the resources it created, which also match our naming conventions in our GraphQL API construct:



Implementing a CDK project - Data source

Next, we need to add our data source. Our example will use a DynamoDB table. Inside the stack class, we'll add some code to create a new table:

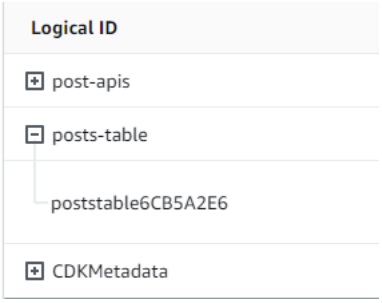
```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // 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
    });
  }
}
```

At this point, let's deploy again:

```
cdk deploy
```

We should check the DynamoDB console for our new table:

Our stack name is correct, and the table name matches our code. If we check our AWS CloudFormation stack again, we'll now see the new table:



Implementing a CDK project - Resolver

This example will use two resolvers: one to query the table and one to add to it. Since we're using pipeline resolvers, we'll need to declare two pipeline resolvers with one function in each. In the query, we'll add the following code:

```
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;
        }
`),
```

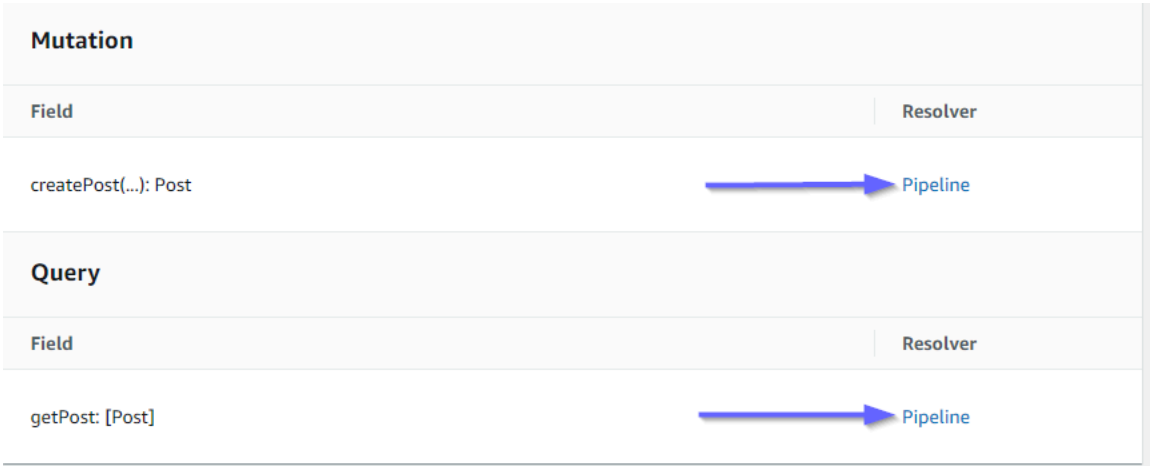
```
      runtime: appsync.FunctionRuntime.JS_1_0_0,
      pipelineConfig: [add_func],
    });

    // 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
    });
  }
}
```

In this snippet, we added a pipeline resolver called `pipeline-resolver-create-posts` with a function called `func-add-post` attached to it. This is the code that will add `Posts` to the table. The other pipeline resolver was called `pipeline-resolver-get-posts` with a function called `func-get-post` that retrieves `Posts` added to the table.

We'll deploy this to add it to the AWS AppSync service:

```
cdk deploy
```

Let's check the AWS AppSync console to see if they were attached to our GraphQL API:



It appears to be correct. In the code, both of these resolvers were attached to the GraphQL API we made (denoted by the `api` props value present in both the resolvers and functions). In the GraphQL API, the fields we attached our resolvers to were also specified in the props (defined by the `typename` and `fieldname` props in each resolver).

Let's see if the content of the resolvers is correct starting with the `pipeline-resolver-get-posts`:

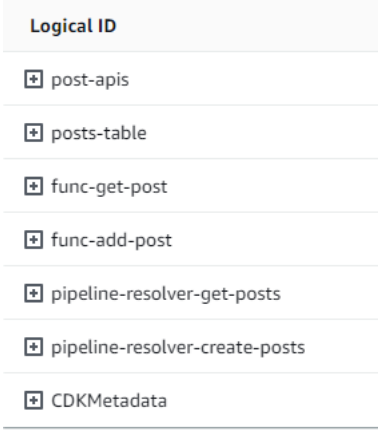
The before and after handlers match our `code` props value. We can also see that a function called `add_posts_func_1`, which matches the name of the function we attached in the resolver.

Let's look at the code content of that function:

This matches up with the `code` props of the `add_posts_func_1` function. Our query was successfully uploaded, so let's check on the query:

These also match the code. If we look at `get_posts_func_1`:

Everything appears to be in place. To confirm this from a metadata perspective, we can check our stack in AWS CloudFormation again:



Now, we need to test this code by performing some requests.

Implementing a CDK project - Requests

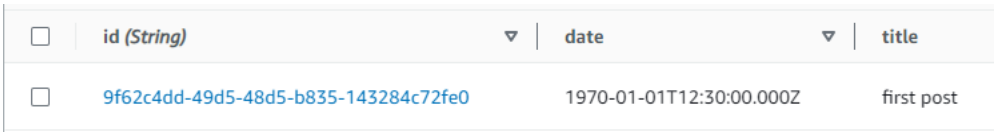
To test our app in the AWS AppSync console, we made one query and one mutation:

```
query MyQuery {
  getPost {
    id
    date
    title
  }
}

mutation MyMutation {
  createPost(input: {date: "1970-01-01T12:30:00.000Z", title: "first post"}) {
    date
    id
    title
  }
}

```

MyMutation contains a createPost operation with the arguments 1970-01-01T12:30:00.000Z and first post. It returns the date and title that we passed in as well as the automatically generated id value. Running the mutation yields the result:

```
{
  "data": {
    "createPost": {
      "date": "1970-01-01T12:30:00.000Z",
      "id": "4dc1c2dd-0aa3-4055-9eca-7c140062ada2",
      "title": "first post"
    }
  }
}
```

If we check the DynamoDB table quickly, we can see our entry in the table when we scan it:



id (String)	date	title
9f62c4dd-49d5-48d5-b835-143284c72fe0	1970-01-01T12:30:00.000Z	first post

Back in the AWS AppSync console, if we run the query to retrieve this Post, we get the following result:

```
{
  "data": {
    "getPost": [
      {
        "id": "9f62c4dd-49d5-48d5-b835-143284c72fe0",
        "date": "1970-01-01T12:30:00.000Z",
        "title": "first post"
```

```
      }
    ]
  }
}
```

Using subscriptions for real-time data applications in AWS AppSync

Important

As of Mar 13, 2025, you can build a real-time PubSub API powered by WebSockets using AWS AppSync Events. For more information, see Publish events via WebSocket in the *AWS AppSync Events Developer Guide*.

AWS AppSync allows you to utilize subscriptions to implement live application updates, push notifications, etc. When clients invoke the GraphQL subscription operations, a secure WebSocket connection is automatically established and maintained by AWS AppSync. Applications can then distribute data in real-time from a data source to subscribers while AWS AppSync continually manages the application's connection and scaling requirements. The following sections will show you how subscriptions in AWS AppSync work.

GraphQL schema subscription directives

Subscriptions in AWS AppSync are invoked as a response to a mutation. This means that you can make any data source in AWS AppSync real time by specifying a GraphQL schema directive on a mutation.

The AWS Amplify client libraries automatically handle subscription connection management. The libraries use pure WebSockets as the network protocol between the client and service.

Note

To control authorization at connection time to a subscription, you can use AWS Identity and Access Management (IAM), AWS Lambda, Amazon Cognito identity pools, or Amazon Cognito user pools for field-level authorization. For fine-grained access controls on subscriptions, you can attach resolvers to your subscription fields and perform logic

using the identity of the caller and AWS AppSync data sources. For more information, see Configuring authorization and authentication to secure your GraphQL APIs.

Subscriptions are triggered from mutations and the mutation selection set is sent to subscribers.

The following example shows how to work with GraphQL subscriptions. It doesn't specify a data source because the data source could be Lambda, Amazon DynamoDB, or Amazon OpenSearch Service.

To get started with subscriptions, you must add a subscription entry point to your schema as follows:

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

Suppose you have a blog post site, and you want to subscribe to new blogs and changes to existing blogs. To do this, add the following `Subscription` definition to your schema:

```
type Subscription {
    addedPost: Post
    updatedPost: Post
    deletedPost: Post
}
```

Suppose further that you have the following mutations:

```
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!
}
```

You can make these fields real time by adding an `@aws_subscribe(mutations: ["mutation_field_1", "mutation_field_2"])` directive for each of the subscriptions you want to receive notifications for, as follows:

```
type Subscription {
    addedPost: Post
    @aws_subscribe(mutations: ["addPost"])
    updatedPost: Post
    @aws_subscribe(mutations: ["updatePost"])
    deletedPost: Post
    @aws_subscribe(mutations: ["deletePost"])
}
```

Because the `@aws_subscribe(mutations: ["",..,""])` takes an array of mutation inputs, you can specify multiple mutations, which initiate a subscription. If you're subscribing from a client, your GraphQL query might look like the following:

```
subscription NewPostSub {
    addedPost {
        __typename
        version
        title
        content
        author
        url
    }
}
```

This subscription query is needed for client connections and tooling.

With the pure WebSockets client, selection set filtering is done per client, as each client can define its own selection set. In this case, the subscription selection set must be a subset of the mutation selection set. For example, a subscription `addedPost{author title}` linked to the mutation `addPost(...){id author title url version}` receives only the author and title of the post. It does not receive the other fields. However, if the mutation lacked the author in its selection set, the subscriber would get a `null` value for the author field (or an error in case the author field is defined as required/not-null in the schema).

The subscription selection set is essential when using pure WebSockets. If a field is not explicitly defined in the subscription, then AWS AppSync doesn't return the field.

In the previous example, the subscriptions didn't have arguments. Suppose that your schema looks like the following:

```
type Subscription {
```

```
    updatedPost(id:ID! author:String): Post
    @aws_subscribe(mutations: ["updatePost"])
}
```

In this case, your client defines a subscription as follows:

```
subscription UpdatedPostSub {
    updatedPost(id:"XYZ", author:"ABC") {
        title
        content
    }
}
```

The return type of a `subscription` field in your schema must match the return type of the corresponding mutation field. In the previous example, this was shown as both `addPost` and `addedPost` returned as a type of `Post`.

To set up subscriptions on the client, see Building a client application using Amplify client.

Using subscription arguments

An important part of using GraphQL subscriptions is understanding when and how to use arguments. You can make subtle changes to modify how and when to notify clients about mutations that have occurred. To do this, see the sample schema from the quickstart chapter, which creates "Todos". For this sample schema, the following mutations are defined:

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

In the default sample, clients can subscribe to updates to any `Todo` by using the `onUpdateTodo` `subscription` with no arguments:

```
subscription OnUpdateTodo {
  onUpdateTodo {
    description
    id
    name
    when
```

```
  }
}
```

You can filter your `subscription` by using its arguments. For example, to only trigger a `subscription` when a `todo` with a specific `ID` is updated, specify the `ID` value:

```
subscription OnUpdateTodo {
  onUpdateTodo(id: "a-todo-id") {
    description
    id
    name
    when
  }
}
```

You can also pass multiple arguments. For example, the following `subscription` demonstrates how to get notified of any `Todo` updates at a specific place and time:

```
subscription todosAtHome {
  onUpdateTodo(when: "tomorrow", where: "at home") {
    description
    id
    name
    when
    where
  }
}
```

Note that all of the arguments are optional. If you don't specify any arguments in your `subscription`, you will be subscribed to all `Todo` updates that occur in your application. However, you could update your `subscription`'s field definition to require the `ID` argument. This would force the response of a specific `todo` instead of all `todos`:

```
onUpdateTodo(
  id: ID!,
  name: String,
  when: String,
  where: String,
  description: String
): Todo
```

Argument null value has meaning

When making a subscription query in AWS AppSync, a `null` argument value will filter the results differently than omitting the argument entirely.

Let's go back to the todos API sample where we could create todos. See the sample schema from the quickstart chapter.

Let's modify our schema to include a new `owner` field, on the `Todo` type, that describes who the owner is. The `owner` field is not required and can only be set on `UpdateTodoInput`. See the following simplified version of the schema:

```
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"])
}
```

The following subscription returns all Todoupdates:

```
subscription MySubscription {
  onUpdateTodo {
    description
    id
    name
    when
    where
  }
}
```

If you modify the preceding subscription to add the field argument `owner: null`, you are now asking a different question. This subscription now registers the client to get notified of all the Todo updates that have not provided an owner.

```
subscription MySubscription {
  onUpdateTodo(owner: null) {
    description
    id
    name
    when
    where
  }
}
```

Note

As of January 1, 2022, MQTT over WebSockets is no longer available as a protocol for GraphQL subscriptions in AWS AppSync APIs. Pure WebSockets is the only protocol supported in AWS AppSync.

Clients based on the AWS AppSync SDK or the Amplify libraries, released after November 2019, automatically use pure WebSockets by default. Upgrading the clients to the latest version allows them to use AWS AppSync's pure WebSockets engine.

Pure WebSockets come with a larger payload size (240 KB), a wider variety of client options, and improved CloudWatch metrics. For more information on using pure WebSocket clients, see Building a real-time WebSocket client in AWS AppSync.

Creating generic pub/sub APIs powered by serverless WebSockets in AWS AppSync

Important

As of Mar 13, 2025, you can build a real-time PubSub API powered by WebSockets using AWS AppSync Events. For more information, see Publish events via WebSocket in the *AWS AppSync Events Developer Guide*.

Some applications only require simple WebSocket APIs where clients listen to a specific channel or topic. Generic JSON data with no specific shape or strongly typed requirements can be pushed to clients listening to one of these channels in a pure and simple publish-subscribe (pub/sub) pattern.

Use AWS AppSync to implement simple pub/sub WebSocket APIs with little to no GraphQL knowledge in minutes by automatically generating GraphQL code on both the API backend and the client sides.

Create and configure pub-sub APIs

To get started, do the following:

1. Sign in to the AWS Management Console and open the AppSync console.
   - In the **Dashboard**, choose **Create API**.
2. On the next screen, choose **Create a real-time API**, then choose **Next**.
3. Enter a friendly name for your pub/sub API.
4. You can enable private API features, but we recommend keeping this off for now. Choose **Next**.
5. You can choose to automatically generate a working pub/sub API using WebSockets. We recommend keeping this feature off for now as well. Choose **Next**.
6. Choose **Create API** and then wait for a couple of minutes. A new pre-configured AWS AppSync pub/sub API will be created in your AWS account.

The API uses AWS AppSync's built-in local resolvers (for more information about using local resolvers, see Tutorial: Local Resolvers in the *AWS AppSync Developer Guide*) to manage multiple temporary pub/sub channels and WebSocket connections, which automatically delivers and filters data to subscribed clients based only on the channel name. API calls are authorized with an API key.

After the API is deployed, you are presented with a couple of extra steps to generate client code and integrate it with your client application. For an example on how to quickly integrate a client, this guide will use a simple React web application.

1. Start by creating a boilerplate React app using NPM on your local machine:

```
$ npx create-react-app mypubsub-app
$ cd mypubsub-app
```

> **Note**
>
> This example uses the Amplify libraries to connect clients to the backend API. However there's no need to create an Amplify CLI project locally. While React is the client of choice in this example, Amplify libraries also support iOS, Android, and Flutter clients, providing the same capabilities in these different runtimes. The supported Amplify clients provide simple abstractions to interact with AWS AppSync GraphQL API backends with few lines of code including built-in WebSocket capabilities fully compatible with the AWS AppSync real-time WebSocket protocol:
>
> ```
> $ npm install @aws-amplify/api
> ```

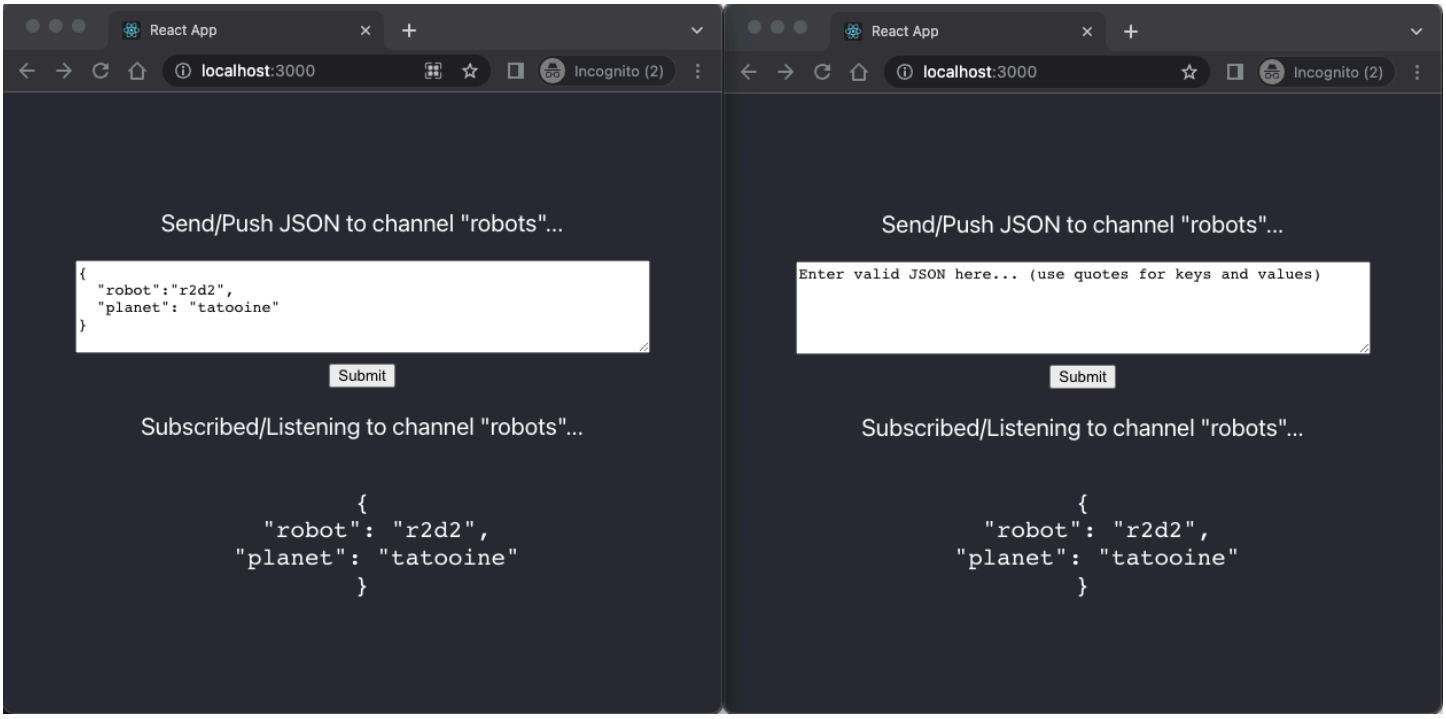
2. In the AWS AppSync console, select **JavaScript**, then **Download** to download a single file with the API configuration details and generated GraphQL operations code.
3. Copy the downloaded file to the `/src` folder in your React project.
4. Next, replace the content of the existing boilerplate `src/App.js` file with the sample client code available in the console.
5. Use the following command to start the application locally:

```
$ npm start
```

6. To test sending and receiving real-time data, open two browser windows and access *localhost:3000*. The sample application is configured to send generic JSON data to a hard-coded channel named *robots*.
7. In one of the browser windows, enter the following JSON blob in the text box then click **Submit**:

```
{
  "robot":"r2d2",
  "planet": "tatooine"
}
```

Both browser instances are subscribed to the *robots* channel and receive the published data in real time, displayed at the bottom of the web application:



All necessary GraphQL API code, including the schema, resolvers, and operations are automatically generated to enable a generic pub/sub use case. On the backend, data is published to AWS AppSync's real-time endpoint with a GraphQL mutation such as the following:

```
mutation PublishData {
    publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
        data
        name
```

```
    }
}
```

Subscribers access the published data sent to the specific temporary channel with a related GraphQL subscription:

```
subscription SubscribeToData {
    subscribe(name:"channel") {
        name
        data
    }
}
```

Implementing pub-sub APIs into existing applications

In case you just need to implement a real-time feature in an existing application, this generic pub/sub API configuration can be easily integrated into any application or API technology. While there are advantages in using a single API endpoint to securely access, manipulate, and combine data from one or more data sources in a single network call with GraphQL, there's no need to convert or rebuild an existing REST-based application from scratch in order to take advantage of AWS AppSync's real-time capabilities. For instance, you could have an existing CRUD workload in a separate API endpoint with clients sending and receiving messages or events from the existing application to the generic pub/sub API for real-time and pub/sub purposes only.

Defining enhanced subscriptions filters in AWS AppSync

> **Important**
>
> As of Mar 13, 2025, you can build a real-time PubSub API powered by WebSockets using AWS AppSync Events. For more information, see Publish events via WebSocket in the *AWS AppSync Events Developer Guide*.

In AWS AppSync, you can define and enable business logic for data filtering on the backend directly in the GraphQL API subscription resolvers by using filters that support additional logical operators. You can configure these filters, unlike the subscription arguments that are defined on the subscription query in the client. For more information about using subscription arguments, see Using subscription arguments. For a list of operators, see AWS AppSync resolver mapping template utility reference.

For the purpose of this document, we divide real-time data filtering into the following categories:

- **Basic filtering** - Filtering based on client-defined arguments in the subscription query.
- **Enhanced filtering** - Filtering based on logic defined centrally in the AWS AppSync service backend.

The following sections explain how to configure enhanced subscription filters and show their practical use.

Defining subscriptions in your GraphQL schema

To use enhanced subscription filters, you define the subscription in the GraphQL schema then define the enhanced filter using a filtering extension. To illustrate how enhanced subscription filtering works in AWS AppSync, use the following GraphQL schema, which defines a ticket management system API, as an example:

```
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
```

Suppose you create a NONE data source for your API, then attach a resolver to the `createTicket` mutation using this data source. Your handlers may look like this:

```
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**
>
> Enhanced filters are enabled in the GraphQL resolver's handler in a given subscription. For more information, see Resolver reference.

To implement the behavior of the enhanced filter, you must use the `extensions.setSubscriptionFilter()` function to define a filter expression evaluated against published data from a GraphQL mutation that the subscribed clients might be interested in. For more information about the filtering extensions, see Extensions.

The following section explains how to use filtering extensions to implement enhanced filters.

Creating enhanced subscription filters using filtering extensions

Enhanced filters are written in JSON in the response handler of the subscription's resolvers. Filters can be grouped together in a list called a `filterGroup`. Filters are defined using at least one rule, each with fields, operators, and values. Let's define a new resolver for `onSpecialTicketCreated` that sets up an enhanced filter. You can configure multiple rules in a filter that are evaluated using AND logic, while multiple filters in a filter group are evaluated using OR logic:

```
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;
}
```

Based on the filters defined in the preceding example, important tickets are automatically pushed to subscribed API clients if a ticket is created with:

- `priority` level `high` or `medium`

  AND

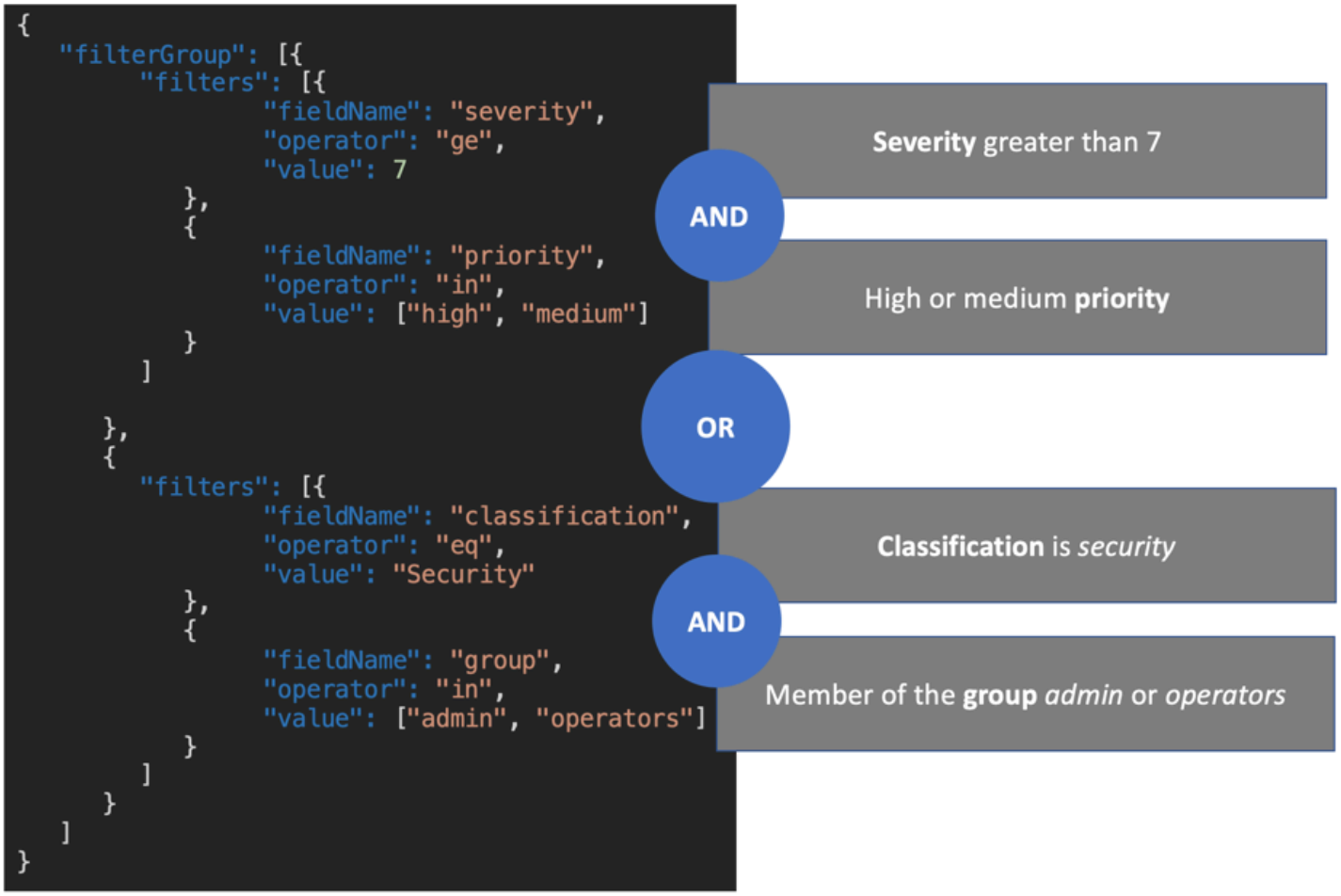
- `severity` level greater than or equal to 7 (ge)

OR

- `classification` ticket set to `Security`

  AND

- `group` assignment set to `admin` or `operators`



Filters defined in the subscription resolver (enhanced filtering) take precedence over filtering based only on subscription arguments (basic filtering). For more information about using subscription arguments, see Using subscription arguments).

If an argument is defined and required in the GraphQL schema of the subscription, filtering based on the given argument takes place only if the argument is defined as a rule in the resolver's `extensions.setSubscriptionFilter()` method. However, if there are no `extensions` filtering methods in the subscription resolver, arguments defined in the client are used only for basic filtering. You can't use basic filtering and enhanced filtering concurrently.

You can use the `context` variable in the subscription's filter extension logic to access contextual information about the request. For example, when using Amazon Cognito User Pools, OIDC, or Lambda custom authorizers for authorization, you can retrieve information about your users in `context.identity` when the subscription is established. You can use that information to establish filters based on your users' identity.

Now assume that you want to implement the enhanced filter behavior for `onGroupTicketCreated`. The `onGroupTicketCreated` subscription requires a mandatory `group` name as an argument. When created, tickets are automatically assigned a `pending` status. You can set up a subscription filter to only receive newly created tickets that belong to the provided group:

```
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;
}
```

When data is published using a mutation like in the following example:

```
mutation CreateTicket {
  createTicket(input: {priority: medium, severity: 2, group: "aws"}) {
    id
```

```
    priority
    severity
    status
    group
    createdAt
  }
}
```

Subscribed clients listen for the data to be automatically pushed via WebSockets as soon as a ticket is created with the `createTicket` mutation:

```
subscription OnGroup {
  onGroupTicketCreated(group: "aws") {
    category
    status
    severity
    priority
    id
    group
    createdAt
    content
  }
}
```

Clients can be subscribed without arguments because the filtering logic is implemented in the AWS AppSync service with enhanced filtering, which simplifies the client code. Clients receive data only if the defined filter criteria is met.

Defining enhanced filters for nested schema fields

You can use enhanced subscription filtering to filter nested schema fields. Suppose we modified the schema from the previous section to include location and address types:

```
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
```

With this schema, you can use a `.` separator to represent nesting. The following example adds a filter rule for a nested schema field under `location.address.country`. The subscription will be triggered if the ticket's address is set to USA:

```
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;
}
```

In the example above, `location` represents nesting level one, `address` represents nesting level two, and `country` represents nesting level three, all of which are separated by the `.` separator.

You can test this subscription by using the `createTicket` mutation:

```
mutation CreateTicketInUSA {
  createTicket(input: {location: "{\"address\":{\"country\":\"USA\"}}"}) {
    category
    content
    createdAt
    group
    id
    location {
      address {
        country
      }
    }
    priority
    severity
    status
  }
}
```

Defining enhanced filters from the client

You can use basic filtering in GraphQL with subscriptions arguments. The client that makes the call in the subscription query defines the arguments' values. When enhanced filters are enabled in an AWS AppSync subscription resolver with the `extensions` filtering, backend filters defined in the resolver take precedence and priority.

Configure dynamic, client-defined enhanced filters using a `filter` argument in the subscription. When you configure these filters, you must update the GraphQL schema to reflect the new argument:

```
...
type Subscription {
    onSpecialTicketCreated(filter: String): Ticket
        @aws_subscribe(mutations: ["createTicket"])
}
...
```

The client can then send a subscription query like in the following example:

```
subscription onSpecialTicketCreated($filter: String) {
     onSpecialTicketCreated(filter: $filter) {
        id
        group
        description
        priority
        severity
     }
 }
```

You can configure the query variable like the following example:

```
{"filter" : "{\"severity\":{\"le\":2}}"}
```

The `util.transform.toSubscriptionFilter()` resolver utility can be implemented in the subscription response mapping template to apply the filter defined in the subscription argument for each client:

```
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;
}
```

With this strategy, clients can define their own filters that use enhanced filtering logic and additional operators. Filters are assigned when a given client invokes the subscription query in a secure WebSocket connection. For more information about the transform utility for enhanced filtering, including the format of the `filter` query variable payload, see JavaScript resolvers overview.

Additional enhanced filtering restrictions

Below are several use cases where additional restrictions are placed on enhanced filters:

- Enhanced filters don't support filtering for top-level object lists. In this use case, published data from the mutation will be ignored for enhanced subscriptions.
- AWS AppSync supports up to five levels of nesting. Filters on schema fields past nesting level five will be ignored. Take the GraphQL response below. The `continent` field in `venue.address.country.metadata.continent` is allowed because it's a level five nest. However, `financial` in `venue.address.country.metadata.capital.financial` is a level six nest, so the filter won't work:

```
{
    "data": {
        "onCreateFilterEvent": {
            "venue": {
                "address": {
                    "country": {
                        "metadata": {
                            "capital": {
                                "financial": "New York"
                            },
                            "continent" : "North America"
                        }
                    },
```

```
                    "state": "WA"
                },
                "builtYear": 2023
            },
            "private": false,
        }
    }
}
```

Unsubscribing WebSocket connections using filters in AWS AppSync

> **Important**
>
> As of Mar 13, 2025, you can build a real-time PubSub API powered by WebSockets using AWS AppSync Events. For more information, see Publish events via WebSocket in the *AWS AppSync Events Developer Guide*.

In AWS AppSync, you can forcibly unsubscribe and close (invalidate) a WebSocket connection from a connected client based on specific filtering logic. This is useful in authorization-related scenarios such as when you remove a user from a group.

Subscription invalidation occurs in response to a payload defined in a mutation. We recommend that you treat mutations used to invalidate subscription connections as administrative operations in your API and scope permissions accordingly by limiting their use to an admin user, group, or backend service. For example, using schema authorization directives such as `@aws_auth(cognito_groups: ["Administrators"])` or `@aws_iam`. For more information, see Using additional authorization modes.

Invalidation filters use the same syntax and logic as enhanced subscription filters. Define these filters using the following utilities:

- `extensions.invalidateSubscriptions()` – Defined in the GraphQL resolver's response handler for a mutation.
- `extensions.setSubscriptionInvalidationFilter()` – Defined in the GraphQL resolver's response handler of the subscriptions linked to the mutation.

For more information about invalidation filtering extensions, see JavaScript resolvers overview.

Using subscription invalidation

To see how subscription invalidation works in AWS AppSync, use the following GraphQL schema:

```
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
}
```

Define an invalidation filter in the `removeUserFromGroup` mutation resolver code:

```
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 };
}
```

When the mutation is invoked, the data defined in the `payload` object is used to unsubscribe the subscription defined in `subscriptionField`. An invalidation filter is also defined in the `onGroupMessageCreated` subscription's response mapping template.

If the `extensions.invalidateSubscriptions()` payload contains an ID that matches the IDs from the subscribed client as defined in the filter, the corresponding subscription is unsubscribed. In addition, the WebSocket connection is closed. Define the subscription resolver code for the `onGroupMessageCreated` subscription:

```
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;
}
```

Note that the subscription response handler can have both subscription filters and invalidation filters defined at the same time.

For example, assume that client A subscribes a new user with the ID *`user-1`* to the group with the ID *`group-1`* using the following subscription request:

```
onGroupMessageCreated(userId : "user-1", groupId: :"group-1"){...}
```

AWS AppSync runs the subscription resolver, which generates subscription and invalidation filters as defined in the preceding `onGroupMessageCreated` response mapping template. For client A, the subscription filters allow data to be sent only to *group-1*, and the invalidation filters are defined for both *user-1* and *group-1*.

Now assume that client B subscribes a user with the ID *user-2* to a group with the ID *group-2* using the following subscription request:

```
onGroupMessageCreated(userId : "user-2", groupId: :"group-2"){...}
```

AWS AppSync runs the subscription resolver, which generates subscription and invalidation filters. For client B, the subscription filters allow data to be sent only to *group-2*, and the invalidation filters are defined for both *user-2* and *group-2*.

Next, assume that a new group message with the ID *message-1* is created using a mutation request like in the following example:

```
createGroupMessage(id: "message-1", groupId :
      "group-1", message: "test message"){...}
```

Subscribed clients matching the defined filters automatically receive the following data payload via WebSockets:

```
{
  "data": {
    "onGroupMessageCreated": {
      "id": "message-1",
      "groupId": "group-1",
      "message": "test message",
    }
  }
}
```

Client A receives the message because the filtering criteria match the defined subscription filter. However, client B doesn't receive the message, as the user is not part of *group-1*. Also, the request doesn't match the subscription filter defined in the subscription resolver.

Finally, assume that *user-1* is removed from *group-1* using the following mutation request:

```
removeUserFromGroup(userId: "user-1", groupId : "group-1"){...}
```

The mutation initiates a subscription invalidation as defined in its `extensions.invalidateSubscriptions()` resolver response handler code. AWS AppSync then unsubscribes client A and closes its WebSocket connection. Client B is unaffected, as the invalidation payload defined in the mutation doesn't match its user or group.

When AWS AppSync invalidates a connection, the client receives a message confirming that they are unsubscribed:

```
{
  "message": "Subscription complete."
}
```

Using context variables in subscription invalidation filters

As with enhanced subscription filters, you can use the `context` variable in the subscription invalidation filter extension to access certain data.

For example, it's possible to configure an email address as the invalidation payload in the mutation, then match it against the email attribute or claim from a subscribed user authorized with Amazon Cognito user pools or OpenID Connect. The invalidation filter defined in the `extensions.setSubscriptionInvalidationFilter()` subscription invalidator checks if the email address set by the mutation's `extensions.invalidateSubscriptions()` payload matches the email address retrieved from the user's JWT token in `context.identity.claims.email`, initiating the invalidation.

Building a real-time WebSocket client in AWS AppSync

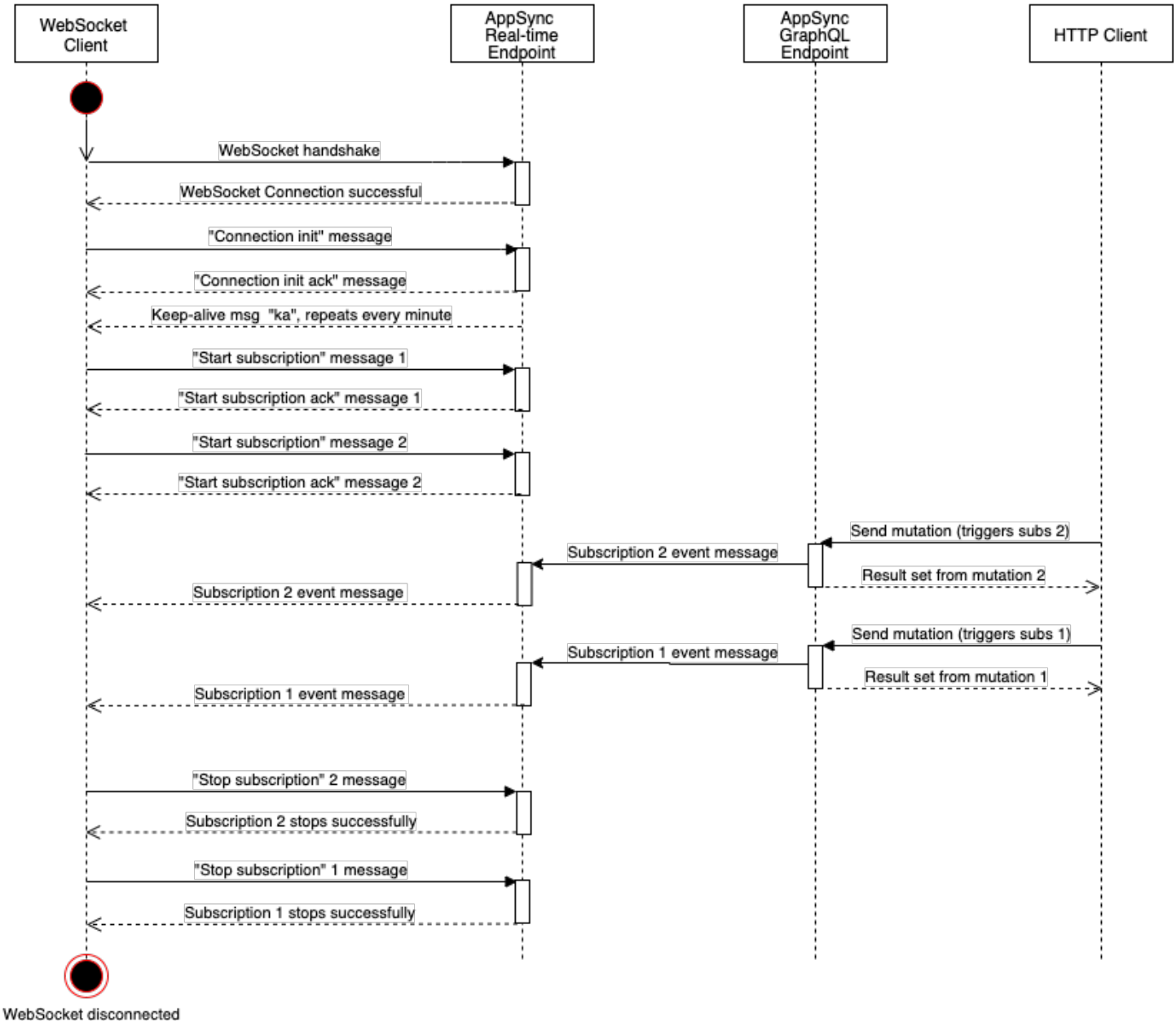
> **Important**
>
> As of Mar 13, 2025, you can build a real-time PubSub API powered by WebSockets using AWS AppSync Events. For more information, see Publish events via WebSocket in the *AWS AppSync Events Developer Guide*..

AWS AppSync's real-time WebSocket client enables GraphQL subscriptions through a multi-step process. The client first establishes a WebSocket connection with the AWS AppSync real-

time endpoint, sends a connection initialization message, and waits for acknowledgment. After successful connection, the client registers subscriptions by sending start messages with unique IDs and GraphQL queries. AWS AppSync confirms successful subscriptions with acknowledgment messages. The client then listens for subscription events, which are triggered by corresponding mutations. To maintain the connection, AWS AppSync sends periodic keep-alive messages. When finished, the client unregisters subscriptions by sending stop messages. This system supports multiple subscriptions on a single WebSocket connection and accommodates various authorization modes, including API keys, Amazon Cognito user pools, IAM, and Lambda.

Real-time WebSocket client implementation for GraphQL subscriptions

The following sequence diagram and steps show the real-time subscriptions workflow between the WebSocket client, HTTP client, and AWS AppSync.



1. The client establishes a WebSocket connection with the AWS AppSync real-time endpoint. If there is a network error, the client should do a jittered exponential backoff. For more information, see Exponential backoff and jitter on the AWS Architecture Blog.
2. (Optional) After successfully establishing the WebSocket connection, the client sends a `connection_init` message.
3. If `connection_init` is sent, the client waits for a `connection_ack` message from AWS AppSync. This message includes a `connectionTimeoutMs` parameter, which is the maximum wait time in milliseconds for a `"ka"` (keep-alive) message.

4. AWS AppSync sends `"ka"` messages periodically. The client keeps track of the time that it received each `"ka"` message. If the client doesn't receive a `"ka"` message within `connectionTimeoutMs` milliseconds, the client should close the connection.
5. The client registers the subscription by sending a `start` subscription message. A single WebSocket connection supports multiple subscriptions, even if they are in different authorization modes.
6. The client waits for AWS AppSync to send `start_ack` messages to confirm successful subscriptions. If there is an error, AWS AppSync returns a `"type": "error"` message.
7. The client listens for subscription events, which are sent after a corresponding mutation is called. Queries and mutations are usually sent through `https://` to the AWS AppSync GraphQL endpoint. Subscriptions flow through the AWS AppSync real-time endpoint using the secure WebSocket (`wss://`).
8. The client unregisters the subscription by sending a `stop` subscription message.
9. After unregistering all subscriptions and checking that there are no messages transferring through the WebSocket, the client can disconnect from the WebSocket connection.

Handshake details to establish the WebSocket connection

To connect and initiate a successful handshake with AWS AppSync, a WebSocket client needs the following:

- The AWS AppSync real-time endpoint
- Headers – Contain information relevant to the AWS AppSync endpoint and authorization. AWS AppSync supports the following three methods for providing headers:
  - Headers via query string
    - The header information is encoded as a base64 string, derived from a stringified JSON object. This JSON object contains details relevant to the AWS AppSync endpoint and authorization. The content of the JSON object varies depending on the authorization mode.
  - Headers via `Sec-WebSocket-Protocol`
    - A base64Url-encoded string from the stringified JSON object that contains information relevant to the AWS AppSync endpoint and authorization is passed as the protocol in the `Sec-WebSocket-Protocol` header. The content of the JSON object varies depending on the authorization mode.
  - Headers via standard HTTP headers:

    - Headers can be passed as standard HTTP headers in the connection request, similar to how headers are passed for GraphQL queries and mutations to AWS AppSync. However, passing headers via standard HTTP headers is not supported for private API connection requests.
- `payload` – Base64-encoded string of `payload`. Payload is needed only if headers are provided using query string

With these requirements, a WebSocket client can connect to the URL, which contains the real-time endpoint with the query string, using `graphql-ws` as the WebSocket protocol.

Discovering the real-time endpoint from the GraphQL endpoint

The AWS AppSync GraphQL endpoint and the AWS AppSync real-time endpoint are slightly different in protocol and domain. You can retrieve the GraphQL endpoint using the AWS Command Line Interface (AWS CLI) command `aws appsync get-graphql-api`.

AWS AppSync GraphQL endpoint:

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync real-time endpoint:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/
graphql
```

Applications can connect to the AWS AppSync GraphQL endpoint (`https://`) using any HTTP client for queries and mutations. Applications can connect to the AWS AppSync real-time endpoint (`wss://`) using any WebSocket client for subscriptions.

With custom domain names, you can interact with both endpoints using a single domain. For example, if you configure `api.example.com` as your custom domain, you can interact with your GraphQL and real-time endpoints using these URLs:

AWS AppSync custom domain GraphQL endpoint:

```
https://api.example.com/graphql
```

AWS AppSync custom domain real-time endpoint:

```
wss://api.example.com/graphql/realtime
```

Header parameter format based on AWS AppSync API authorization mode

The format of the `header` object used in the connection query string varies depending on the AWS AppSync API authorization mode. The `host` field in the object refers to the AWS AppSync GraphQL endpoint, which is used to validate the connection even if the `wss://` call is made against the real-time endpoint. To initiate the handshake and establish the authorized connection, the `payload` should be an empty JSON object. Payload is needed only if headers are passed via query string.

The following sections demonstrate the header formats for each authorization mode.

API key

API key header

Header contents

- `"host": <string>`: The host for the AWS AppSync GraphQL endpoint or your custom domain name.
- `"x-api-key": <string>`: The API key configured for the AWS AppSync API.

Example

```
{
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com",
    "x-api-key":"da2-12345678901234567890123456"
}
```

Headers via query string

First, a JSON object containing the `host` and the `x-api-key` is converted into a string. Next, this string is encoded using base64 encoding. The resulting base64-encoded string is added as a query parameter named `header` to the WebSocket URL for establishing the connection with the AWS AppSync real-time endpoint. The resulting request URL takes the following form:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJob3N0IjoiZXhhbXBsZTEyMzQ1Njc4OTAwMDAuYXBwc3luYy1hcGkudXMtZWFzdC0xLmFtYXpvbmF3cy5jb20i
```

It's important to note that in addition to the base64-encoded header object, an empty JSON object {} is also base64-encoded and included as a separate query parameter named `payload` in the WebSocket URL.

Headers via `Sec-WebSocket-Protocol`

A JSON object containing the `host` and the `x-api-key` is converted to a string and then encoded using base64Url encoding. The resulting base64Url-encoded string is prefixed with `header-`. This prefixed string is then used as a new sub-protocol in addition to `graphql-ws` in the `Sec-WebSocket-Protocol` header when establishing the WebSocket connection with the AWS AppSync real-time endpoint.

The resulting request URL takes the following form:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

The `Sec-WebSocket-Protocol` header contains the following value:

```
"sec-websocket-protocol" : ["graphql-ws", "header-
ewogICAgImhvc3QiOiJleGFtcGxlMTIzNDU2Nzg5MDAwMC5hcHBzeW5jLWFwaS51cy1lYXN0LTEuYW1hem9uYXdzLmNvbSI
```

Headers via standard HTTP headers

In this method, the host and API key information is transmitted using standard HTTP headers when establishing the WebSocket connection with the AWS AppSync real-time endpoint. The resulting request URL takes the following form:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

The request headers would include the following:

```
"sec-websocket-protocol" : ["graphql-ws"]
"host":"example1234567890000.appsync-api.us-east-1.amazonaws.com",
"x-api-key":"da2-12345678901234567890123456"
```

Amazon Cognito user pools and OpenID Connect (OIDC)

Amazon Cognito and OIDCheader

Header contents:

- `"Authorization": <string>`: A JWT ID token. The header can use a Bearer scheme.

- `"host": <string>`: The host for the AWS AppSync GraphQL endpoint or your custom domain name.

Example:

```
{

 "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"
}
```

Headers via query string

First, a JSON object containing the `host` and the `Authorization` is converted into a string. Next, this string is encoded using base64 encoding. The resulting base64-encoded string is added as a query parameter named `header` to the WebSocket URL for establishing the connection with the AWS AppSync real-time endpoint. The resulting request URL takes the following form:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

It's important to note that in addition to the base64-encoded header object, an empty JSON object {} is also base64-encoded and included as a separate query parameter named `payload` in the WebSocket URL.

Headers via `Sec-WebSocket-Protocol`

A JSON object containing the `host` and the `Authorization` is converted to a string and then encoded using base64Url encoding. The resulting base64Url-encoded string is prefixed with `header-`. This prefixed string is then used as a new sub-protocol in addition to `graphql-ws` in the `Sec-WebSocket-Protocol` header when establishing the WebSocket connection with the AWS AppSync real-time endpoint.

The resulting request URL takes the following form:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

The `Sec-WebSocket-Protocol` header contains the following value:

```
"sec-websocket-protocol" : ["graphql-ws", "header-
ewogICAgImhvc3QiOiJleGFtcGxlMTIzNDU2Nzg5MDAwMC5hcHBzeW5jLWFwaS51cy1lYXN0LTEuYW1hem9uYXdzLmNvbSI
```

Headers via standard HTTP headers

In this method, the host and Authorization information is transmitted using standard HTTP headers when establishing the WebSocket connection with the AWS AppSync real-time endpoint. The resulting request URL takes the following form:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

The request headers would include the following:

```
"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 header

Header content

- `"accept": "application/json, text/javascript"`: A constant <string> parameter.
- `"content-encoding": "amz-1.0"`: A constant <string> parameter.
- `"content-type": "application/json; charset=UTF-8"`: A constant <string> parameter.
- `"host": <string>`: This is the host for the AWS AppSync GraphQL endpoint.
  - `"x-amz-date": <string>`: The timestamp must be in UTC and in the following ISO 8601 format: YYYYMMDD'T'HHMMSS'Z'. For example, 20150830T123600Z is a valid timestamp.

Do not include milliseconds in the timestamp. For more information, see Handling dates in Signature Version 4 in the *AWS General Reference*.

- `"X-Amz-Security-Token": <string>`: The AWS session token, which is required when using temporary security credentials. For more information, see Using temporary credentials with AWS resources in the *IAM User Guide*.
- `"Authorization": <string>`: Signature Version 4 (SigV4) signing information for the AWS AppSync endpoint. For more information on the signing process, see Task 4: Add the signature to the HTTP request in the *AWS General Reference*.

The SigV4 signing HTTP request includes a canonical URL, which is the AWS AppSync GraphQL endpoint with `/connect` appended. The service endpoint AWS Region is same Region where you're using the AWS AppSync API, and the service name is 'appsync'. The HTTP request to sign is the following:

```
{
  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",
  }
}
```

Example

```
{
  "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"
}
```

Headers via query string

First, a JSON object containing the `host` (AWS AppSync GraphQL endpoint) and the other authorization headers is converted to a string. Next, this string is encoded using base64 encoding. The resulting base64-encoded string is added to the WebSocket URL as a query parameter named `header`. The resulting request URL takes the following form:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

It's important to note that in addition to the base64-encoded header object, an empty JSON object {} is also base64-encoded and included as a separate query parameter named `payload` in the WebSocket URL.

Headers via `Sec-WebSocket-Protocol`

A JSON object containing the `host` and the other authorization headers is converted to a string and then encoded using base64Url encoding. The resulting base64Url-encoded string is prefixed with `header-`. This prefixed string is then used as a new sub-protocol in addition to `graphql-ws` in the `Sec-WebSocket-Protocol` header when establishing the WebSocket connection with the AWS AppSync real-time endpoint.

The resulting request URL takes the following form:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

The `Sec-WebSocket-Protocol` header contains the following value:

```
"sec-websocket-protocol" : ["graphql-ws", "header-
ew0KICAiYWNjZXB0IjogImFwcGxpY2F0aW9uL2pzb24sIHRleHQvamF2YXNjcmlwdCIsDQogICJjb250ZW50LWVuY29kaW5
```

Headers via standard HTTP headers

In this method, the host and the other authorization information is transmitted using standard HTTP headers when establishing the WebSocket connection with the AWS AppSync real-time endpoint. The resulting request URL takes the following form:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

The request headers would include the following:

```
"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"
```

To sign the request using a custom domain:

```
{
  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",
  }
}
```

Example

```
{
  "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=XXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
 Signature=83EXAMPLEbcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}
```

Request URL with query string

```
wss://api.example.com/graphql?
header=eyEXAMPLEHQiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoE
```

> **Note**
>
> One WebSocket connection can have multiple subscriptions (even with different authentication modes). One way to implement this is to create a WebSocket connection for the first subscription and then close it when the last subscription is unregistered. You can optimize this by waiting a few seconds before closing the WebSocket connection, in case the app is subscribed immediately after the last subscription is unregistered. For a mobile app example, when changing from one screen to another, on *unmounting* event it stops a subscription, and on *mounting* event it starts a different subscription.

Lambda authorization

Lambda authorization header

Header content

- `"Authorization": <string>`: The value that is passed as `authorizationToken`.
- `"host": <string>`: The host for the AWS AppSync GraphQL endpoint or your custom domain name.

Example

```
{

 "Authorization":"M0UzQzM1MkQtMkI0Ni00OTZCLUI1NkQtMUM0MTQ0QjVBRTczCkI1REEzRTIxLTk5NzItNDJENi1BQ
```

```
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

Headers via query string

First, a JSON object containing the `host` and the `Authorization` is converted into a string. Next, this string is encoded using base64 encoding. The resulting base64-encoded string is added as a query parameter named `header` to the WebSocket URL for establishing the connection with the AWS AppSync real-time endpoint. The resulting request URL takes the following form:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

It's important to note that in addition to the base64-encoded header object, an empty JSON object {} is also base64-encoded and included as a separate query parameter named `payload` in the WebSocket URL.

Headers via `Sec-WebSocket-Protocol`

A JSON object containing the `host` and the `Authorization` is converted to a string and then encoded using base64Url encoding. The resulting base64Url-encoded string is prefixed with `header-`. This prefixed string is then used as a new sub-protocol in addition to `graphql-ws` in the `Sec-WebSocket-Protocol` header when establishing the WebSocket connection with the AWS AppSync real-time endpoint.

The resulting request URL takes the following form:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

The `Sec-WebSocket-Protocol` header contains the following value:

```
"sec-websocket-protocol" : ["graphql-ws", "header-
ewogICAgImhvc3QiOiJleGFtcGxlMTIzNDU2Nzg5MDAwMC5hcHBzeW5jLWFwaS51cy1lYXN0LTEuYW1hem9uYXdzLmNvbSI
```

Headers via standard HTTP headers

In this method, the host and Authorization information is transmitted using standard HTTP headers when establishing the WebSocket connection with the AWS AppSync real-time endpoint. The resulting request URL takes the following form:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

The request headers would include the following:

```
"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"
```

Real-time WebSocket operation

After initiating a successful WebSocket handshake with AWS AppSync, the client must send a subsequent message to connect to AWS AppSync for different operations. These messages require the following data:

- `type`: The type of the operation.
- `id`: A unique identifier for the subscription. We recommend using a UUID for this purpose.
- `payload`: The associated payload, depending on the operation type.

The `type` field is the only required field; the `id` and `payload` fields are optional.

Sequence of events

To successfully initiate, establish, register, and process the subscription request, the client must step through the following sequence:

1. Initialize connection (`connection_init`)
2. Connection acknowledgment (`connection_ack`)
3. Subscription registration (`start`)
4. Subscription acknowledgment (`start_ack`)
5. Processing subscription (`data`)
6. Subscription unregistration (`stop`)

Connection init message

(Optional) After a successful handshake, the client can send the `connection_init` message to start communicating with the AWS AppSync real-time endpoint. The message is a string obtained by stringifying the JSON object as follows:

```
{ "type": "connection_init" }
```

Connection acknowledge message

After sending the `connection_init` message, the client must wait for the `connection_ack` message. All messages sent before receiving `connection_ack` are ignored. The message should read as follows:

```
{
  "type": "connection_ack",
  "payload": {
    // Time in milliseconds waiting for ka message before the client should terminate
 the WebSocket connection
    "connectionTimeoutMs": 300000
  }
}
```

Keep-alive message

In addition to the connection acknowledgment message, the client periodically receives keep-alive messages. If the client doesn't receive a keep-alive message within the connection timeout period, the client should close the connection. AWS AppSync keeps sending these messages and servicing the registered subscriptions until it shuts down the connection automatically (after 24 hours). Keep-alive messages are heartbeats and do not need the client to acknowledge them.

```
{ "type": "ka" }
```

Subscription registration message

After the client receives a `connection_ack` message, the client can send subscription registration messages to AWS AppSync. This type of message is a stringified JSON object that contains the following fields:

- `"id": <string>`: The ID of the subscription. This ID must be unique for each subscription, otherwise the server returns an error indicating that the subscription ID is duplicated.
- `"type": "start"`: A constant `<string>` parameter.
- `"payload": <Object>`: An object that contains the information relevant to the subscription.
  - `"data": <string>`: A stringified JSON object that contains a GraphQL query and variables.
    - `"query": <string>`: A GraphQL operation.
    - `"variables": <Object>`: An object that contains the variables for the query.
  - `"extensions": <Object>`: An object that contains an authorization object.
- `"authorization": <Object>`: An object that contains the fields required for authorization.

Authorization object for subscription registration

The same rules in the Header parameter format based on AWS AppSync API authorization mode section apply for the authorization object. The only exception is for IAM, where the SigV4 signature information is slightly different. For more details, see the IAM example.

Example using Amazon Cognito user pools:

```
{
  "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"
}
```

Example using IAM:

```
{
```

```
  "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"
}
```

Example using a custom domain name:

```
{
```

```
  "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"
}
```

The SigV4 signature does not need `/connect` to be appended to the URL, and the JSON stringified GraphQL operation replaces `data`. The following is an example of a SigV4 signature request:

```
{
  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",
  }
}
```

Subscription acknowledgment message

After sending the subscription start message, the client should wait for AWS AppSync to send the `start_ack` message. The `start_ack` message indicates that the subscription is successful.

Subscription acknowledgment example:

```
{
  "type": "start_ack",
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

Error message

If connection init or subscription registration fails, or if a subscription is ended from the server, the server sends an error message to the client. If the error happens during connection init time, the connection will be closed by the server.

- `"type": "error"`: A constant `<string>` parameter.
- `"id": <string>`: The ID of the corresponding registered subscription, if relevant.
- `"payload" <Object>`: An object that contains the corresponding error information.

Example:

```
{
  "type": "error",
  "payload": {
    "errors": [
      {
        "errorType": "LimitExceededError",
        "message": "Rate limit exceeded"
      }
    ]
  }
}
```

Processing data messages

When a client submits a mutation, AWS AppSync identifies all of the subscribers interested in it and sends a `"type":"data"` message to each using the corresponding subscription `id` from the `"start"` subscription operation. The client is expected to keep track of the subscription `id` that it sends so that when it receives a data message, the client can match it with the corresponding subscription.

- `"type": "data"`: A constant `<string>` parameter.
- `"id": <string>`: The ID of the corresponding registered subscription.
- `"payload" <Object>`: An object that contains the subscription information.

Example:

```
{
  "type": "data",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": {
      "onCreateMessage": {
        "__typename": "Message",
        "message": "test"
      }
    }
  }
}
```

Subscription unregistration message

When the app wants to stop listening to the subscription events, the client should send a message with the following stringified JSON object:

- `"type": "stop"`: A constant `<string>` parameter.
- `"id": <string>`: The ID of the subscription to unregister.

Example:

```
{
  "type":"stop",
  "id":"ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

AWS AppSync sends back a confirmation message with the following stringified JSON object:

- `"type": "complete"`: A constant `<string>` parameter.
- `"id": <string>`: The ID of the unregistered subscription.

After the client receives the confirmation message, it receives no more messages for this particular subscription.

Example:

```
{
```

```
  "type":"complete",
  "id":"eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

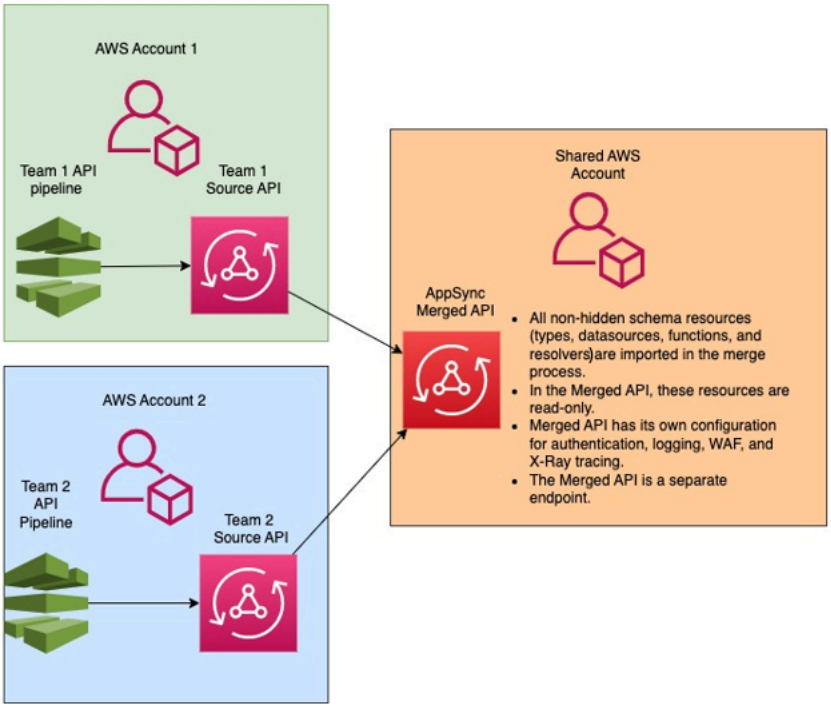
Disconnecting the WebSocket

Before disconnecting, to avoid data loss, the client should have the necessary logic to check that no operation is currently in place through the WebSocket connection. All subscriptions should be unregistered before disconnecting from the WebSocket.

Merging APIs in AWS AppSync

As the use of GraphQL expands within an organization, trade-offs between API ease-of-use and API development velocity can arise. One the one hand, organizations adopt AWS AppSync and GraphQL to simplify application development by giving developers a flexible API they can use to securely access, manipulate, and combine data from one or more data domains with a single network call. On the other hand, teams within an organization that are responsible for the different data domains combined into a single GraphQL API endpoint may want the ability to create, manage, and deploy API updates independent of each other in order to increase their development velocities.

To resolve this tension, the AWS AppSync Merged APIs feature allows teams from different data domains to independently create and deploy AWS AppSync APIs (e.g., GraphQL schemas, resolvers, data sources, and functions), that can then be combined into a single, merged API. This gives organizations the ability to maintain a simple to use, cross domain API, and a way for the different teams that contribute to that API the ability to quickly and independently make API updates.



Using Merged APIs, organizations can import the resources of multiple, independent source AWS AppSync APIs into a single AWS AppSyncMerged API endpoint. To do this, AWS AppSync allows you to create a list of source AWS AppSync source APIs, and then merge all of the metadata associated with the source APIs including schema, types, datasources, resolvers, and functions, into a new AWS AppSync merged API.

During merges, there's the possibility that a merge conflict will occur due to inconsistencies in the source API data content such as type naming conflicts when combining multiple schemas. For simple use cases where no definitions in the source APIs conflict, there's no need to modify the source API schemas. The resulting Merged API simply imports all types, resolvers, data sources and functions from the original source AWS AppSync APIs. For complex use cases where conflicts arise, the users/teams will have to resolve the conflicts through various means. AWS AppSync provides users with several tools and examples that can reduce merge conflicts.

Subsequent merges that are configured in AWS AppSync will propagate changes made in the source APIs to the associated Merged API.

Merged APIs and Federation

There are many solutions and patterns in the GraphQL community for combining GraphQL schemas and enabling team collaboration through a shared graph. AWS AppSync Merged APIs adopt a *build time* approach to schema composition, where source APIs are combined into a separate, Merged API. An alternative approach is to layer a *run time* router across multiple source APIs or sub-graphs. In this approach, the router receives a request, references a combined schema that it maintains as metadata, constructs a request plan, and then distributes request elements across its underlying sub-graphs/servers.The following table compares the AWS AppSync Merged API build-time approach with router-based, run-time approaches to GraphQL schema composition:

Feature	AppSync Merged API	Router-based solutions
Sub-graphs managed independently	Yes	Yes
Sub-graphs addressable independently	Yes	Yes
Automated schema compositi on	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/deserializati on, 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 associate d 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 Merged APIs can only be associated with AWS AppSync source APIs. If you need support for schema composition across AWS AppSync and non-AWS AppSync sub-graphs, you can connect one or more AWS AppSync GraphQL and/or Merged APIs into a router-based solution. For example, see the reference blog for adding AWS AppSync APIs as a sub-graph using a router-based architecture with Apollo Federation v2: Apollo GraphQL Federation with AWS AppSync.

Topics

Merged API conflict resolution

In the event of a merge conflict, AWS AppSync provides users with several tools and examples to help troubleshoot the issue(s).

Merged API schema directives

AWS AppSync has introduced several GraphQL directives that can be used to- reduce or resolve conflicts across source APIs:

- *@canonical*: This directive sets the precedence of types/fields with similar names and data. If two or more source APIs have the same GraphQL type or field, one of the APIs can annotate their type or field as *canonical*, which will be prioritized during the merge. Conflicting types/fields that aren't annotated with this directive in other source APIs are ignored when merged.
- *@hidden*: This directive encapsulates certain types/fields to remove it from the merging process. Teams may want to remove or hide specific types or operations in the source API so only internal clients can access specific typed data. With this directive attached, types or fields are not merged into the Merged API.
- *@renamed*: This directive changes the names of types/fields to reduce naming conflicts. There are situations where different APIs have the same type or field name. However, they all need to be available in the merged schema. A simple way to include them all in the Merged API is to rename the field to something similar but different.

To show the utility schema directives provide, consider the following example:

In this example, let's assume that we want to merge two source APIs. We're given two schemas that create and retrieve posts (e.g., comment section or social media posts). Assuming that the types and fields are very similar, there's a high chance for conflict during a merge operation. The snippets below show the types and fields of each schema.

The first file, called *Source1.graphql*, is a GraphQL schema that allows a user to create `Posts` using the `putPost` mutation. Each `Post` contains a title and an ID. The ID is used to reference the `User`, or poster's information (email and address), and the `Message`, or the payload (content). The `User` type is annotated with the *@canonical* tag.

```
# 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
}
```

The second file, called *Source2.graphql*, is a GraphQL schema that does very similar things as *Source1.graphql*. However, notice that the fields of each type are different. When merging these two schemas, there will be merge conflicts because of these differences.

Also note how *Source2.graphql* also contains several directives to reduce these conflicts. The `Post` type is annotated with a *@hidden* tag to obfuscate itself during the merge operation. The `Message` type is annotated with the *@renamed* tag to modify the type name to `ChatMessage` in the event of a naming conflict with another `Message` type.

```
# 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")
}
```

When the merge occurs, the result will produce the `MergedSchema.graphql` file:

```
# 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
}
```

Several things occurred in the merge:

- The `User` type from *Source1.graphql* was prioritized over the `User` from *Source2.graphql* due to the *@canonical* annotation.
- The `Message` type from *Source1.graphql* was included in the merge. However, the `Message` from *Source2.graphql* had a naming conflict. Due to its *@renamed* annotation, it was also included in the merge but with the alternative name `ChatMessage`.
- The `Post` type from *Source1.graphql* was included, but the `Post` type from *Source2.graphql* wasn't. Normally, there would be a conflict on this type, but because the `Post` type from *Source2.graphql* had a *@hidden* annotation, its data was obfuscated and not included in the merge. This resulted in no conflicts.
- The `Query` type was updated to include the contents from both files. However, one `GetMessage` query was renamed to `GetChatMessage` due to the directive. This resolved the naming conflict between the two queries with the same name.

There's also the case of no directives being added to a conflicting type. Here, the merged type will include the union of all fields from all source definitions of that type. For instance, consider the following example:

This schema, called *Source1.graphql*, allows for creating and retrieving `Posts`. The configuration is similar to the previous example, but with less information.

```
# 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
}
```

This schema, called *Source2.graphql*, allows for creating and retrieving `Reviews` (e.g., movie rating or restaurant reviews). `Reviews` are associated with the `Post` of the same ID value. Together, they contain the title, post ID, and payload message of the full review post.

When merging, there will be a conflict between the two `Post` types. Because there are no annotations to resolve this issue, the default behavior is to perform a union operation on the conflicting types.

```
# 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
}
```

When the merge occurs, the result will produce the `MergedSchema.graphql` file:

```
# 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
}
```

Several things occurred in the merge:

- The `Mutation` type faced no conflicts and was merged.
- The `Post` type fields were combined via union operation. Notice how the union between the two produced a single `id`, a `title`, and a single `reviews`.
- The `Review` type faced no conflicts and was merged.
- The `Query` type faced no conflicts and was merged.

Managing resolvers on shared types

In the above example, consider the case where *Source1.graphql* has configured a unit resolver on `Query.getPost`, which uses a DynamoDB data source named `PostDatasource`. This resolver will return the `id` and `title` of a `Post` type. Now, consider *Source2.graphql* has configured a pipeline resolver on `Post.reviews`, which runs two functions. `Function1` has a `None` data source attached to perform custom authorization checks. `Function2` has a DynamoDB data source attached to query the `reviews` table.

```
query GetPostQuery {
    getPost(id: "1") {
        id,
```

```
        title,
        reviews
    }
}
```

When the query above is run by a client to the Merged API endpoint, the AWS AppSync service first runs the unit resolver for `Query.getPost` from `Source1`, which calls the `PostDatasource` and returns the data from DynamoDB. Then, it runs the `Post.reviews` pipeline resolver in which `Function1` performs custom authorization logic and `Function2` returns the reviews given the `id` found in `$context.source`. The service processes the request as a single GraphQL run, and this simple request will only require a single request token.

Managing resolver conflicts on shared types

Consider the following case where we also implement a resolver on `Query.getPost` in order to provide multiple fields at a time beyond the field resolver in `Source2`. *Source1.graphql* may look like this:

```
# This snippet represents a file called Source1.graphql

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

type Query {
    getPost(id: ID!): Post
}
```

Source2.graphql may look like this:

```
# This snippet represents a file called Source2.graphql

type Post  {
  id: ID!
  content: String!
  contentHash: String!
  author: String!
}

type Query {
```

```
    getPost(id: ID!): Post
}
```

Attempting to merge these two schemas will generate a merge error because AWS AppSync Merged APIs don't allow multiple source resolvers to be attached to the same field. In order to resolve this conflict, you can implement a field resolver pattern that would require *Source2.graphql* to add a separate type that will define the fields that it owns from the `Post` type. In the following example, we add a type called `PostInfo`, which contains the content and author fields that will be resolved by *Source2.graphql*. *Source1.graphql* will implement the resolver attached to `Query.getPost`, while *Source2.graphql* will now attach a resolver to `Post.postInfo`to ensure that all data can be successfully retrieved:

```
type Post  {
  id: ID!
  postInfo: PostInfo
}

type PostInfo {
   content: String!
   contentHash: String!
   author: String!
}

type Query {
    getPost(id: ID!): Post
}
```

While resolving such a conflict requires source API schemas to be rewritten and, potentially, clients to change their queries, the advantage of this approach is that ownership of merged resolvers remains clear across source teams.

Configuring schemas

Two parties are responsible for configuring the schemas to create a Merged API:

- **Merged API owners** - Merged API owners must configure the Merged API's authorization logic and advanced settings like logging, tracing, caching, and WAF support.
- **Associated source API owners** - Associated API owners must configure the schemas, resolvers, and datasources that make up the Merged API.

Because your Merged API's schema is created from the schemas of your associated source APIs, it's **read only**. This means changes to the schema must be initiated in your source APIs. In the AWS AppSync console, you can toggle between your Merged schema and the individual schemas of the source APIs included in your Merged API using the drop-down list above the **Schema** window.

Configuring authorization modes

Multiple authorization modes are available to protect your Merged API. To learn more about authorization modes in AWS AppSync, see Authorization and authentication.

The following authorization modes are available to use with Merged APIs:

- **API key**: The simplest authorization strategy. All requests must include an API key under the `x-api-key` request header. Expired API keys are kept for 60 days after the expiration date.
- **AWS Identity and Access Management (IAM)**: The AWS IAM authorization strategy authorizes all requests that are **sigv4 signed**.
- **Amazon Cognito User Pools**: Authorize your users via Amazon Cognito User Pools to achieve more fine-grained control.
- **AWS Lambda Authorizers**: A serverless function that allows you to authenticate and authorize access to your AWS AppSync API using custom logic.
- **OpenID Connect**: This authorization type enforces OpenID connect (OIDC) tokens provided by an OIDC-compliant service. Your application can leverage users and privileges defined by your OIDC provider for controlling access.

The authorization modes of a Merged API are configured by the Merged API owner. At the time of a merge operation, the Merged API must include the primary authorization mode configured on a source API either as its own primary authorization mode or as a secondary authorization mode. Otherwise, it will be incompatible, and the merge operation will fail with a conflict. When using multi-auth directives in the source APIs, the merging process is able to automatically merge these directives into the unified endpoint. In the case where the primary authorization mode of the source API doesn't match the primary authorization mode of the Merged API, it will automatically add these auth directives to ensure that the authorization mode for the types in the source API is consistent.

Configuring execution roles

When you create a Merged API, you need to define a service role. An AWS service role is an AWS Identity and Access Management (IAM) role that is used by AWS services to perform tasks on your behalf.

In this context, it's necessary for your Merged API to run resolvers that access data from the data sources configured in your source APIs. The required service role for this is the `mergedApiExecutionRole`, and it must have explicit access to run requests on source APIs included in your merged API via the `appsync:SourceGraphQL` IAM permission. During the run of a GraphQL request, the AWS AppSync service will assume this service role and authorize the role to perform the `appsync:SourceGraphQL` action.

AWS AppSync supports allowing or denying this permission on specific top-level fields within the request like how the IAM authorization mode works for IAM APIs. For non-top-level fields, AWS AppSync requires you to define the permission on the source API ARN itself. In order to restrict access to specific non-top-level fields in the Merged API, we recommend implementing custom logic within your Lambda or hiding the source API fields from the Merged API using the *@hidden* directive. If you want to allow the role to perform all data operations within a source API, you can add the policy below. Note that the first resource entry allows access to all top-level fields and the second entry covers child resolvers that authorize on the source API resource itself:

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"]
    }]
}
```

If you want to limit the access to only a specific top-level field, you can use a policy like this:

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"]
    }]
}
```

You can also use the AWS AppSync console API creation wizard to generate a service role to allow your Merged API to access resources configured in source APIs that are in the same account as your merged API. In the case where your source APIs are not in the same account as your merged API, you must first share your resources using AWS Resource Access Manager (AWS RAM).

Configuring cross-account Merged APIs using AWS RAM

When you create a Merged API, you can optionally associate source APIs from other accounts that have been shared via AWS Resource Access Manager (AWS RAM). AWS RAM helps you share your resources securely across AWS accounts, within your organization or organizational units (OUs), and with IAM roles and users.

AWS AppSync integrates with AWS RAM in order to support configuring and accessing source APIs across multiple accounts from a single Merged API. AWS RAM allows you to create a resource share, or a container of resources and the permission sets that will be shared for each of them. You can add AWS AppSync APIs to a resource share in AWS RAM. Within a resource share, AWS AppSync provides three different permission sets that can be associated with an AWS AppSync API in RAM:

1. `AWSRAMPermissionAppSyncSourceApiOperationAccess`: The default permission set that's added when sharing an AWS AppSync API in AWS RAM if no other permission is specified. This permission set is used for sharing a source AWS AppSync API with a Merged API owner. This permission set includes the permission for `appsync:AssociateMergedGraphqlApi` on the

source API as well as the `appsync:SourceGraphQL` permission required to access the source API resources at runtime.

2. `AWSRAMPermissionAppSyncMergedApiOperationAccess`: This permission set should be configured when sharing a Merged API with a source API owner. This permission set will give the source API the ability to configure the Merged API including the ability to associate any source APIs owned by the target principal to the Merged API and to read and update the source API associations of the Merged API.
3. `AWSRAMPermissionAppSyncAllowSourceGraphQLAccess`: This permission set allows the `appsync:SourceGraphQL` permission to be used with an AWS AppSync API. It is intended to be used for sharing a source API with a Merged API owner. In contrast to the default permission set for source API operation access, this permission set only includes the runtime permission `appsync:SourceGraphQL`. If a user opts to share the Merged API operation access to a source API owner, they will also need to share this permission from the source API to the Merged API owner in order to have runtime access through the Merged API endpoint.

AWS AppSync also supports customer-managed permissions. When one of the provided AWS-managed permissions doesn't work, you can create your own customer-managed permission. Customer-managed permissions are managed permissions that you author and maintain by precisely specifying which actions can be performed under which conditions with resources shared using AWS RAM. AWS AppSync allows you to choose from the following actions when creating your own permission:

1. `appsync:AssociateSourceGraphqlApi`
2. `appsync:AssociateMergedGraphqlApi`
3. `appsync:GetSourceApiAssociation`
4. `appsync:UpdateSourceApiAssociation`
5. `appsync:StartSchemaMerge`
6. `appsync:ListTypesByAssociation`
7. `appsync:SourceGraphQL`

Once you have properly shared a source API or Merged API in AWS RAM and, if necessary, the resource share invitation has been accepted, it will be visible in the AWS AppSync console when you create or update the source API associations on your Merged API. You can also list all AWS AppSync APIs that have been shared using AWS RAM with your account regardless of the

permission set by calling the `ListGraphqlApis` operation provided by AWS AppSync and using the `OTHER_ACCOUNTS` owner filter.

> **Note**
>
> Sharing via AWS RAM requires the caller in AWS RAM to have permission to perform the `appsync:PutResourcePolicy` action on any API that is being shared.

Merging

Managing merges

Merged APIs are meant to support team collaboration on a unified AWS AppSync endpoint. Teams can independently evolve their own isolated source GraphQL APIs in the backend while the AWS AppSync service manages the integration of the resources into the single Merged API endpoint in order to reduce friction in collaboration and decrease development lead times.

Auto-merges

Source APIs associated with your AWS AppSync Merged API can be configured to automatically merge (auto-merge) into the Merged API after any changes are made to the source API. This ensures that the changes from the source API are always propagated to the Merged API endpoint in the background. Any change in the source API schema will be updated in the Merged API so long as it does not introduce a merge conflict with an existing definition in the Merged API. If the update in the source API is updating a resolver, data source, or function, the imported resource will also be updated.When a new conflict is introduced that cannot be automatically resolved (auto-resolved), the Merged API schema update is rejected due to an unsupported conflict during the merge operation. The error message is available in the console for each source API association that has a status of `MERGE_FAILED`. You can also inspect the error message by calling the `GetSourceApiAssociation` operation for a given source API association using the AWS SDK or using the AWS CLI like so:

```
aws appsync get-source-api-association --merged-api-identifier <Merged API ARN> --
association-id <SourceApiAssociation id>
```

This will produce a result in the following format:

```
{
```

```
    "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."
    }
}
```

Manual merges

The default setting for a source API is a manual merge. To merge any changes that have occurred in the source APIs since the Merged API was last updated, the source API owner can invoke a manual merge from the AWS AppSync console or via the `StartSchemaMerge` operation available in the AWS SDK and AWS CLI.

Additional support for Merged APIs

Configuring subscriptions

Unlike router-based approaches to GraphQL schema composition, AWS AppSync Merged APIs provide built-in support for GraphQL subscriptions. All subscription operations defined in your associated source APIs will automatically merge and function in your Merged API without modification. To learn more about how AWS AppSync supports subscriptions via serverless WebSockets connection, see Real-time data.

Configuring observability

AWS AppSync Merged APIs provide built-in logging, monitoring and metrics via Amazon CloudWatch. AWS AppSync also provides built-in support for tracing via AWS X-Ray.

Configuring custom domains

AWS AppSync Merged APIs provide built-in support for using custom domains with your Merged API's GraphQL and Real-time endpoints.

Configuring caching

AWS AppSync Merged APIs provide built-in support for optionally caching request-level and/or resolver-level responses as well as response compression. To learn more, see Caching and compression.

Configuring private APIs

AWS AppSync Merged APIs provide built-in support for Private APIs that limit access to your Merged API's GraphQL and Real-time endpoints to traffic originating from VPC endpoints that you can configure.

Configuring firewall rules

AWS AppSync Merged APIs provide built-in support for AWS WAF, which enables you to protect your APIs by defining web application firewall rules.

Configuring audit logs

AWS AppSync Merged APIs provide built-in support for AWS CloudTrail, which enables you to configure and manage audit logs.

Merged API limitations

When developing Merged APIs, take note of the following rules:

1. A Merged API cannot be a source API for another Merged API.
2. A source API cannot be associated with more than one Merged API.
3. The default size limit for a Merged API schema document is 10 MB.
4. The default number of source APIs that can be associated with a Merged API is 10. However, you can request a limit increase if you need more than 10 source APIs in your Merged API.

Creating Merged APIs

To create a Merged API in the console

1. Sign in to the AWS Management Console and open the AWS AppSync console.
   - In the **Dashboard** , choose **Create API**.

2. Choose **Merged API**, then choose **Next**.
3. In the **Specify API details** page, enter in the following information:

   a. Under **API Details**, enter in the following information:

      i. Specify your merged API's **API name**. This field is a way to label your GraphQL API to conveniently distinguish it from other GraphQL APIs.

      ii. Specify the **Contact details**. This field is optional and attaches a name or group to the GraphQL API. It's not linked to or generated by other resources and works much like the API name field.

   b. Under **Service role**, you must attach an IAM execution role to your merged API so that AWS AppSync can securely import and use your resources at runtime. You can choose to **Create and use a new service role**, which will allow you to specify the policies and resources that AWS AppSync will use. You can also import an existing IAM role by choosing **Use an existing service role**, then selecting the role from the drop-down list.

   c. Under **Private API configuration**, you can choose to enable private API features. Note that this choice cannot be changed after creating the merged API. For more information about private APIs, see Using AWS AppSync Private APIs.

   Choose **Next** after you're done.
4. Next, you must add the GraphQL APIs that will be used as the foundation for your merged API. In the **Select source APIs** page, enter in the following information:

   a. In the **APIs from your AWS account** table, choose **Add Source APIs**. In the list of GraphQL APIs, each entry will contain the following data:

      i. **Name**: The GraphQL API's **API name** field.

      ii. **API ID**: The GraphQL API's unique ID value.

      iii. **Primary auth mode**: The default authorization mode for the GraphQL API. For more information about authorization modes in AWS AppSync, see Authorization and authentication.

      iv. **Additonal auth mode**: The secondary authorization modes that were configured in the GraphQL API.

      v. Choose the APIs that you will use in the merged API by selecting the checkbox next to the API's **Name** field. Afterwards, choose **Add Source APIs**. The selected GraphQL APIs will appear in the **APIs from your AWS accounts** table.

b. In the **APIs from other AWS accounts** table, choose **Add Source APIs**. The GraphQL APIs in this list come from other accounts that are sharing their resources to yours through AWS Resource Access Manager (AWS RAM). The process for selecting GraphQL APIs in this table is the same as the process in the previous section. For more information about sharing resources through AWS RAM, see What is AWS Resource Access Manager?.

   Choose **Next** after you're done.

c. Add your primary auth mode. See Authorization and authentication for more information. Choose **Next**.

d. Review your inputs, then choose **Create API**.

Building GraphQL APIs with RDS introspection

AWS AppSync's introspection utility can discover models from database tables and propose GraphQL types. The AWS AppSync console's Create API wizard can instantly generate an API from an Aurora MySQL or PostgreSQL database. It automatically creates types and JavaScript resolvers to read and write data.

AWS AppSync provides direct integration with Amazon Aurora databases through the Amazon RDS Data API. Rather than requiring a persistent database connection, the Amazon RDS Data API offers a secure HTTP endpoint that AWS AppSync connects to for running SQL statements. You can use this to create a relational database API for your MySQL and PostgreSQL workloads on Aurora.

Building an API for your relational database with AWS AppSync has several advantages:

- Your database is not directly exposed to clients, decoupling the access point from the database itself.
- You can build purpose-built APIs tailored to the needs of different applications, removing the need for custom business logic in frontends. This aligns with the Backend-For-Frontend (BFF) pattern.
- Authorization and access control can be implemented at the AWS AppSync layer using various authorization modes to control access. No additional compute resources are required to connect to the database, such as hosting a web server or proxying connections.
- Real-time capabilities can be added via subscriptions, with data mutations made through AppSync automatically pushed to connected clients.
- Clients can connect to the API over HTTPS using common ports like 443.

AWS AppSync makes building APIs from existing relational databases easy. Its introspection utility can discover models from database tables and propose GraphQL types. The AWS AppSync console's *Create API* wizard can instantly generate an API from an Aurora MySQL or PostgreSQL database. It automatically creates types and JavaScript resolvers to read and write data.

AWS AppSync provides integrated JavaScript utilities to simplify writing SQL statements in resolvers. You can use AWS AppSync's `sql` tag templates for static statements with dynamic values, or the `rds` module utilities to build statements programmatically. See the resolver function reference for RDS data sources and built-in modules for more.

Using the introspection feature (console)

For a detailed tutorial and getting started guide, see Tutorial: Aurora PostgreSQL Serverless with Data API.

The AWS AppSync console allows you to create an AWS AppSync GraphQL API from your existing Aurora database configured with the Data API in just a few minutes. This quickly generates an operational schema based on your database configuration. You can use the API as-is or build on it to add features.

1. Sign in to the AWS Management Console and open the AppSync console.
   - In the **Dashboard**, choose **Create API**.
2. Under **API options**, choose **GraphQL APIs**, **Start with an Amazon Aurora cluster**, then **Next**.
   a. Enter an **API name**. This will be used as an identifier for the API in the console.
   b. For **contact details**, you can enter a point of contact to identify a manager for the API. This is an optional field.
   c. Under **Private API configuration**, you can enable private API features. A private API can only be accessed from a configured VPC endpoint (VPCE). For more information, see Private APIs.

      We don't recommend enabling this feature for this example. Choose **Next** after reviewing your inputs.
3. In the **Database** page, choose **Select database**.
   a. You need to choose your database from your cluster. The first step is to choose the **Region** in which your cluster exists.

b. Choose the **Aurora cluster** from the drop-down list. Note that you must have created and enabled a corresponding data API before using the resource.

c. Next, you must add the credentials for your database to the service. This is primarily done using AWS Secrets Manager. Choose the **Region** in which your secret exists. For more information on how to retrieve secret information, see Find secrets or Retrieve secrets.

d. Add your secret from the drop-down list. Note that the user must have read permissions for your database.

4. Choose **Import**.

AWS AppSync will start introspecting your database, discovering tables, columns, primary keys, and indexes. It checks that the discovered tables can be supported in a GraphQL API. Note that to support creating new rows, tables need a primary key, which can use multiple columns. AWS AppSync maps table columns to type fields as follows:

Data type	Field type
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. Once table discovery is complete, the **Database** section will be populated with your information. In the new **Database tables** section, the data from the table may already be populated and converted to a type for your schema. If you don't see some of the required data, you can check for it by choosing **Add tables**, clicking on the checkboxes for those types in the modal that appears, then choosing **Add**.

   To remove a type from the **Database tables** section, click on the checkbox next to the type you want to remove, then choose **Remove**. The removed types will be placed in the **Add tables** modal if you want to add them again later.

   Note that AWS AppSync uses the table names as type names, but you can rename them - for example, changing a plural table name like *`movies`* to the type name *`Movie`*. To rename a type in the **Database tables** section, click on the checkbox of the type you want to rename, then click on the *pencil* icon in the **Type name** column.

   To preview the content of the schema based on your selections, choose **Preview schema**. Note that this schema cannot be empty, so you'll have to have at least one table converted to a type. Also, this schema cannot exceed 1 MB in size.

   - Under **Service role**, choose whether to create a new service role specifically for this import or use an existing role.
6. Choose **Next**.
7. Next, choose whether to create a read-only API (queries only) or an API for reading and writing data (with queries and mutations). The latter also supports real-time subscriptions triggered by mutations.
8. Choose **Next**.
9. Review your choices and then choose **Create API**. AWS AppSync will create the API and attach resolvers to queries and mutations. The generated API is fully operational and can be extended as needed.

Using the introspection feature (API)

You can use the `StartDataSourceIntrospection` introspection API to discover models in your database programmatically. For more details on the command, see using the `StartDataSourceIntrospection` API.

To use `StartDataSourceIntrospection`, provide your Aurora cluster Amazon Resource Name (ARN), database name, and AWS Secrets Manager secret ARN. The command starts the introspection process. You can retrieve the results with the `GetDataSourceIntrospection` command. You can specify whether the command should return the Storage Definition Language (SDL) string for the discovered models. This is useful for generating an SDL schema definition directly from the discovered models.

For example, if you have the following Data definition language (DDL) statement for a simple `Todos` table:

```
create table if not exists public.todos
(
id serial constraint todos_pk primary key,
description text,
due timestamp,
"createdAt" timestamp default now()
);
```

You start the introspection with the following.

```
aws appsync start-data-source-introspection \
  --rds-data-api-config resourceArn=<cluster-arn>,secretArn=<secret-
arn>,databaseName=database
```

Next, use the `GetDataSourceIntrospection` command to retrieve the result.

```
aws appsync get-data-source-introspection \
  --introspection-id a1234567-8910-abcd-efgh-identifier \
  --include-models-sdl
```

This returns the following result.

```
{
    "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
    }
}
```

Building a client application using Amplify client

You can connect to your AWS AppSync GraphQL API using any GraphQL client, but we strongly recommend the Amplify v6 client. Amplify not only autogenerates strongly typed client SDKs for your GraphQL API but also offers support for real-time data and enhanced GraphQL query capabilities in client applications. For web applications, Amplify can produce a JavaScript client. For those targeting cross-platform or mobile environments, Amplify caters to Android, iOS, and React Native. To delve deeper into client code generation for these platforms, consult the Amplify documentation. Here's a guide to kickstart your journey with a JavaScript React application:

> **Note**
>
> You need to install and configure both npm and the Amazon CLI before getting started. If you're using the Amplify v6 client, follow this guide.

To get started:

1. On your local machine, navigate to your project's directory. Install the Amplify library using the command below:

   ```
   npm install aws-amplify
   ```

2. Download your configuration file and place it in your project folder. Your configuration file will typically contain a `config` variable with some settings (endpoint, Region, authorization mode, etc.) defined. For example, it may look like this:

   ```
   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. In your code, import the Amplify Library and your configuration to set up Amplify:

```
import { Amplify } from 'aws-amplify';
import config from './aws-exports.js';

Amplify.configure(config);
```

   Alternatively, use the snippet in your API configuration to set up Amplify directly:

```
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. Using the Amplify toolchain, you have the option to autogenerate operations based on your schema, which saves you the effort of manual scripting. In your application's root directory, use the following CLI command:

```
npx @aws-amplify/cli codegen add --apiId <id goes here> --region <region goes here>
```

   This will download your API's schema and, by default, generate client helper code into the `src/graphql` folder. After every API deployment, you can rerun the following command to generate updated GraphQL statements and types:

```
npx @aws-amplify/cli codegen
```

5. You can now generate models for Android, Swift, Flutter, and JavaScript DataStore. Use the following command to download your schema:

```
aws appsync get-introspection-schema --api-id <id goes here> --region <region goes
 here> --format SDL schema.graphql
```

Then, run the following command from your application's root directory:

```
npx @aws-amplify/cli codegen models \
--model-schema schema.graphql \
--target [android|ios|flutter|javascript|typescript] \
--output-dir ./
```

JavaScript resolver tutorials for AWS AppSync

Data sources and resolvers are used by AWS AppSync to translate GraphQL requests and fetch information from your AWS resources. AWS AppSync supports automatic provisioning and connections with certain data source types. AWS AppSync also supports AWS Lambda, Amazon DynamoDB, relational databases (Amazon Aurora Serverless), Amazon OpenSearch Service, and HTTP endpoints as data sources. You can use a GraphQL API with your existing AWS resources or build data sources and resolvers from scratch. The following sections are meant to elucidate some of the more common GraphQL use cases in the form of tutorials.

Topics

Creating a simple post application using DynamoDB JavaScript resolvers

In this tutorial, you will import your Amazon DynamoDB tables to AWS AppSync and connect them to build a fully-functional GraphQL API using JavaScript pipeline resolvers that you can leverage in your own application.

You will use the AWS AppSync console to provision your Amazon DynamoDB resources, create your resolvers, and connect them to your data sources. You will also be able to read and write to your Amazon DynamoDB database through GraphQL statements and subscribe to real-time data.

There are specific steps that must be completed in order for GraphQL statements to be translated to Amazon DynamoDB operations and for responses to be translated back into GraphQL. This

tutorial outlines the configuration process through several real-world scenarios and data access patterns.

Creating your GraphQL API

To create a GraphQL API in AWS AppSync

1. Open the AppSync console and choose **Create API**.
2. Select **Design from scratch** and choose **Next**.
3. Name your API `PostTutorialAPI`, then choose **Next**. Skip to the review page while keeping the rest of the options set to their default values and choose `Create`.

The AWS AppSync console creates a new GraphQL API for you. By detault, it's using the API key authentication mode. You can use the console to set up the rest of the GraphQL API and run queries against it for the rest of this tutorial.

Defining a basic post API

Now that you have your GraphQL API, you can set up a basic schema that allows the basic creation, retrieval, and deletion of post data.

To add data to your schema

1. In your API, choose the **Schema** tab.
2. We will create a schema that defines a `Post` type and an operation `addPost` to add and get `Post` objects. In the **Schema** pane, replace the contents with the following code:

```
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. Choose **Save Schema**.

Setting up your Amazon DynamoDB table

The AWS AppSync console can help provision the AWS resources needed to store your own resources in an Amazon DynamoDB table. In this step, you'll create an Amazon DynamoDB table to store your posts. You'll also set up a secondary index that we'll use later.

To create your Amazon DynamoDB table

1. On the **Schema** page, choose **Create Resources**.
2. Choose **Use existing type**, then choose the `Post` type.
3. In the **Additional Indexes** section, choose **Add Index**.
4. Name the index `author-index`.
5. Set the `Primary key` to `author` and the `Sort` key to None.
6. Disable **Automatically generate GraphQL**. In this example, we'll create the resolver ourselves.
7. Choose **Create**.

You now have a new data source called `PostTable`, which you can see by visiting **Data sources** in the side tab. You will use this data source to link your queries and mutations to your Amazon DynamoDB table.

Setting up an addPost resolver (Amazon DynamoDB PutItem)

Now that AWS AppSync is aware of the Amazon DynamoDB table, you can link it to individual queries and mutations by defining resolvers. The first resolver you create is the `addPost` pipeline resolver using JavaScript, which enables you to create a post in your Amazon DynamoDB table. A pipeline resolver has the following components:

- The location in the GraphQL schema to attach the resolver. In this case, you are setting up a resolver on the `createPost` field on the `Mutation` type. This resolver will be invoked when the caller calls mutation `{ addPost(...){...} }`.
- The data source to use for this resolver. In this case, you want to use the DynamoDB data source you defined earlier, so you can add entries into the `post-table-for-tutorial` DynamoDB table.
- The request handler. The request handler is a function that handles the incoming request from the caller and translates it into instructions for AWS AppSync to perform against DynamoDB.
- The response handler. The job of the response handler is to handle the response from DynamoDB and translate it back into something that GraphQL expects. This is useful if the shape of the data in DynamoDB is different to the `Post` type in GraphQL, but in this case they have the same shape, so you just pass the data through.

To set up your resolver

1. In your API, choose the **Schema** tab.
2. In the **Resolvers** pane, find the `addPost` field under the `Mutation` type, then choose **Attach**.
3. Choose your data source, then choose **Create**.
4. In your code editor, replace the code with this snippet:

```
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. Choose **Save**.

> **Note**
>
> In this code, you use the DynamoDB module utils that allow you to easily create DynamoDB requests.

AWS AppSync comes with a utility for automatic ID generation called `util.autoId()`, which is used to generate an ID for your new post. If you do not specify an ID, the utility will automatically generate it for you.

```
const key = { id: ctx.args.id ?? util.autoId() }
```

For more information about the utilities available for JavaScript, see JavaScript runtime features for resolvers and functions.

Call the API to add a post

Now that the resolver has been configured, AWS AppSync can translate an incoming `addPost` mutation to an Amazon DynamoDB `PutItem` operation. You can now run a mutation to put something in the table.

To run the operation

1. In your API, choose the **Queries** tab.
2. In the **Queries** pane, add the following mutation:

```
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. Choose **Run** (the orange play button), then choose `addPost`. The results of the newly created post should appear in the **Results** pane to the right of the **Queries** pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

The following explanation shows what occurred:

1. AWS AppSync received an `addPost` mutation request.
2. AWS AppSync executes the request handler of the resolver. The `ddb.put` function creates a `PutItem` request that looks like this:

```
{
  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 uses this value to generate and execute a Amazon DynamoDB `PutItem` request.
4. AWS AppSync took the results of the `PutItem` request and converted them back to GraphQL types.

```
{
    "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. The response handler returns the result immediately (`return ctx.result`).
6. The final result is visible in the GraphQL response.

Setting up the getPost resolver (Amazon DynamoDB GetItem)

Now that you're able to add data to the Amazon DynamoDB table, you need to set up the `getPost` query so it can retrieve that data from the table. To do this, you set up another resolver.

To add your resolver

1. In your API, choose the **Schema** tab.
2. In the **Resolvers** pane on the right, find the `getPost` field on the `Query` type and then choose **Attach**.
3. Choose your data source, then choose **Create**.
4. In the code editor, replace the code with this snippet:

```
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. Save your resolver.

> **Note**
>
> In this resolver, we use an arrow function expression for the response handler.

Call the API to get a post

Now that the resolver has been set up, AWS AppSync knows how to translate an incoming `getPost` query to an Amazon DynamoDB `GetItem` operation. You can now run a query to retrieve the post you created earlier.

To run your query

1. In your API, choose the **Queries** tab.
2. In the **Queries** pane, add the following code, and use the id that you copied after creating your post:

```
query getPost {
  getPost(id: "123") {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. Choose **Run** (the orange play button), then choose `getPost`. The results of the newly created post should appear in the **Results** pane to the right of the **Queries** pane.

4. The post retrieved from Amazon DynamoDB should appear in the **Results** pane to the right of the **Queries** pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

Alternatively, take the following example:

```
query getPost {
  getPost(id: "123") {
    id
    author
    title
  }
}
```

If your `getPost` query only needs the `id`, `author`, and `title`, you can change your request function to use projection expressions to specify only the attributes that you want from your DynamoDB table to avoid unnecessary data transfer from DynamoDB to AWS AppSync. For example, the request function may look like the snippet below:

```
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
```

You can also use a selectionSetList with `getPost` to represent the `expression`:

```
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
```

Create an updatePost mutation (Amazon DynamoDB UpdateItem)

So far, you can create and retrieve `Post` objects in Amazon DynamoDB. Next, you'll set up a new mutation to update an object. Compared to the `addPost` mutation that requires all fields to be specified, this mutation allows you to only specify the fields that you want to change. It also introduced a new `expectedVersion` argument that allows you to specify the version that you want to modify. You'll set up a condition that makes sure that you are modifying the latest version of the object. You'll do this using the `UpdateItem` Amazon DynamoDB operation.sc

To update your resolver

1. In your API, choose the **Schema** tab.
2. In the **Schema** pane, modify the `Mutation` type to add a new `updatePost` mutation as follows:

```
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. Choose **Save Schema**.
4. In the **Resolvers** pane on the right, find the newly created `updatePost` field on the `Mutation` type, then choose **Attach**. Create your new resolver using the snippet below:

```
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. Save any changes you made.

This resolver uses `ddb.update` to create an Amazon DynamoDB `UpdateItem` request. Instead of writing the entire item, you're just asking Amazon DynamoDB to update certain attributes. This is done using Amazon DynamoDB update expressions.

The ddb.update function takes a key and an update object as arguments. Then, you check the values of the incoming arguments. When a value is set to null, use the DynamoDB remove operation to signal that the value should be removed from the DynamoDB item.

There is also a new condition section. A condition expression allows you tell AWS AppSync and Amazon DynamoDB whether or not the request should succeed based on the state of the object already in Amazon DynamoDB before the operation is performed. In this case, you only want the UpdateItem request to succeed if the version field of the item currently in Amazon DynamoDB matches the expectedVersion argument exactly. When the item is updated, we want to increment the value of the version. This is easy to do with the operation function increment.

For more information about condition expressions, see the Condition expressions documentation.

For more info about the UpdateItem request, see the UpdateItem documentation and the DynamoDB module documentation.

For more information about how to write update expressions, see the DynamoDB UpdateExpressions documentation.

Call the API to update a post

Let's try updating the Post object with the new resolver.

To update your object

1. In your API, choose the **Queries** tab.
2. In the **Queries** pane, add the following mutation. You'll also need to update the id argument to the value you noted down earlier:

```
mutation updatePost {
  updatePost(
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
```

```
    ups
    downs
    version
  }
}
```

3. Choose **Run** (the orange play button), then choose `updatePost`.
4. The updated post in Amazon DynamoDB should appear in the **Results** pane to the right of the **Queries** pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

In this request, you asked AWS AppSync and Amazon DynamoDB to update the `title` and `content` fields only. All of the other fields were left alone (other than incrementing the `version` field). You set the `title` attribute to a new value and removed the `content` attribute from the post. The `author`, `url`, `ups`, and `downs` fields were left untouched. Try executing the mutation request again while leaving the request exactly as is. You should see a response similar to the following:

```
{
  "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)"
    }
  ]
}
```

The request fails because the condition expression evaluates to `false`:

1. The first time you ran the request, the value of the `version` field of the post in Amazon DynamoDB was 1, which matched the `expectedVersion` argument. The request succeeded, which meant the `version` field was incremented in Amazon DynamoDB to 2.
2. The second time you ran the request, the value of the `version` field of the post in Amazon DynamoDB was 2, which did not match the `expectedVersion` argument.

This pattern is typically called *optimistic locking*.

Create vote mutations (Amazon DynamoDB UpdateItem)

The `Post` type contains `ups` and `downs` fields to enable the recording of upvotes and downvotes. However, at this moment, the API doesn't let us do anything with them. Let's add a mutation to let us upvote and downvote the posts.

To add your mutation

1. In your API, choose the **Schema** tab.
2. In the **Schema** pane, modify the `Mutation` type and add the `DIRECTION` enum to add new vote mutations:

```
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. Choose **Save Schema**.
4. In the **Resolvers** pane on the right, find the newly created `vote` field on the `Mutation` type, and then choose **Attach**. Create a new resolver by creating and replacing the code with the following snippet:

```
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. Save any changes you made.

Call the API to upvote or downvote a post

Now that the new resolvers have been set up, AWS AppSync knows how to translate an incoming `upvotePost` or `downvote` mutation to an Amazon DynamoDB `UpdateItem` operation. You can now run mutations to upvote or downvote the post you created earlier.

To run your mutation

1. In your API, choose the **Queries** tab.
2. In the **Queries** pane, add the following mutation. You'll also need to update the `id` argument to the value you noted down earlier:

```
mutation votePost {
  vote(id:123, direction: UP) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. Choose **Run** (the orange play button), then choose `votePost`.
4. The updated post in Amazon DynamoDB should appear in the **Results** pane to the right of the **Queries** pane. It should look similar to the following:

```
{
  "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. Choose **Run** a few more times. You should see the `ups` and `version` fields incrementing by `1` each time you execute the query.
6. Change the query to call it with a different `DIRECTION`.

```
mutation votePost {
  vote(id:123, direction: DOWN) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

7. Choose **Run** (the orange play button), then choose `votePost`.

   This time, you should see the `downs` and `version` fields incrementing by `1` each time you run the query.

Setting up a deletePost resolver (Amazon DynamoDB DeleteItem)

Next, you'll want to create a mutation to delete a post. You'll do this using the `DeleteItem` Amazon DynamoDB operation.

To add your mutation

1. In your schema, choose the **Schema** tab.
2. In the **Schema** pane, modify the `Mutation` type to add a new `deletePost` mutation:

```
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. This time, you made the `expectedVersion` field optional. Next, choose **Save Schema**.
4. In the **Resolvers** pane on the right, find the newly created `delete` field in the `Mutation` type, then choose **Attach**. Create a new resolver using the following code:

```
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**
>
> The `expectedVersion` argument is an optional argument. If the caller set an `expectedVersion` argument in the request, the request handler adds a condition that only allows the `DeleteItem` request to succeed if the item is already deleted or if the `version` attribute of the post in Amazon DynamoDB exactly matches the `expectedVersion`. If left out, no condition expression is specified on the `DeleteItem` request. It succeeds regardless of the value of `version` or whether or not the item exists in Amazon DynamoDB.
> Even though you're deleting an item, you can return the item that was deleted, if it was not already deleted.

For more info about the `DeleteItem` request, see the DeleteItem documentation.

Call the API to delete a post

Now that the resolver has been set up, AWS AppSync knows how to translate an incoming `delete` mutation to an Amazon DynamoDB `DeleteItem` operation. You can now run a mutation to delete something in the table.

To run your mutation

1. In your API, choose the **Queries** tab.
2. In the **Queries** pane, add the following mutation. You'll also need to update the `id` argument to the value you noted down earlier:

```
mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. Choose **Run** (the orange play button), then choose `deletePost`.
4. The post is deleted from Amazon DynamoDB. Note that AWS AppSync returns the value of the item that was deleted from Amazon DynamoDB, which should appear in the **Results** pane to the right of the **Queries** pane. It should look similar to the following:

```
{
  "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. The value is only returned if this call to `deletePost` is the one that actually deletes it from Amazon DynamoDB. Choose **Run** again.
6. The call still succeeds, but no value is returned:

```
{
  "data": {
    "deletePost": null
  }
}
```

7. Now, let's try deleting a post, but this time specifying an `expectedValue`. First, you'll need to create a new post because you've just deleted the one you've been working with so far.
8. In the **Queries** pane, add the following mutation:

```
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. Choose **Run** (the orange play button), then choose `addPost`.
10. The results of the newly created post should appear in the **Results** pane to the right of the **Queries** pane. Record the `id` of the newly created object because you'll need it in just a moment. It should look similar to the following:

```
{
  "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. Now, let's try to delete that post with an illegal value for **expectedVersion**. In the **Queries** pane, add the following mutation. You'll also need to update the `id` argument to the value you noted down earlier:

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
```

```
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

12. Choose **Run** (the orange play button), then choose `deletePost`. The following result is returned:

```
{
  "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. The request failed because the condition expression evaluates to `false`. The value for `version` of the post in Amazon DynamoDB doesn't match the `expectedValue` specified in the arguments. The current value of the object is returned in the `data` field in the `errors` section of the GraphQL response. Retry the request, but correct the `expectedVersion`:

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

14. Choose **Run** (the orange play button), then choose `deletePost`.

    This time the request succeeds, and the value that was deleted from Amazon DynamoDB is returned:

```
{
  "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. Choose **Run** again. The call still succeeds, but this time no value is returned because the post was already deleted in Amazon DynamoDB.

```
{ "data": { "deletePost": null } }
```

Setting up an allPost resolver (Amazon DynamoDB Scan)

So far, the API is only useful if you know the `id` of each post you want to look at. Let's add a new resolver that returns all the posts in the table.

To add your mutation

1. In your API, choose the **Schema** tab.
2. In the **Schema** pane, modify the `Query` type to add a new `allPost` query as follows:

```
type Query {
    allPost(limit: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

3. Add a new `PaginationPosts` type:

```
type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}
```

4. Choose **Save Schema**.
5. In the **Resolvers** pane on the right, find the newly created `allPost` field in the `Query` type, then choose **Attach**. Create a new resolver with the following code:

```
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 };
}
```

   This resolver's request handler expects two optional arguments:

- `limit` - Specifies the maximum number of items to return in a single call.

- `nextToken` - Used to retrieve the next set of results (we'll show where the value for `nextToken` comes from later).

6. Save any changes made to your resolver.

For more information about `Scan` request, see the Scan reference documentation.

Call the API to scan all posts

Now that the resolver has been set up, AWS AppSync knows how to translate an incoming `allPost` query to an Amazon DynamoDB `Scan` operation. You can now scan the table to retrieve all the posts. Before you can try it out though, you need to populate the table with some data because you've deleted everything you've worked with so far.

To add and query data

1. In your API, choose the **Queries** tab.
2. In the **Queries** pane, add the following mutation:

```
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. Choose **Run** (the orange play button).

4. Now, let's scan the table, returning five results at a time. In the **Queries** pane, add the following query:

```
query allPost {
  allPost(limit: 5) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

5. Choose **Run** (the orange play button), then choose `allPost`.

   The first five posts should appear in the **Results** pane to the right of the **Queries** pane. It should look similar to the following:

```
{
  "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. You received five results and a `nextToken` that you can use to get the next set of results. Update the `allPost` query to include the `nextToken` from the previous set of results:

```
query allPost {
  allPost(
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
      author
    }
    nextToken
  }
}
```

7. Choose **Run** (the orange play button), then choose `allPost`.

   The remaining four posts should appear in the **Results** pane to the right of the **Queries** pane. There is no `nextToken` in this set of results because you've paged through all nine posts with none remaining. It should look similar to the following:

```
{
  "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
    }
  }
}
```

Setting up an allPostsByAuthor resolver(Amazon DynamoDB Query)

In addition to scanning Amazon DynamoDB for all posts, you can also query Amazon DynamoDB to retrieve posts created by a specific author. The Amazon DynamoDB table you created earlier already has a `GlobalSecondaryIndex` called `author-index` that you can use with an Amazon DynamoDB `Query` operation to retrieve all posts created by a specific author.

To add your query

1. In your API, choose the **Schema** tab.
2. In the **Schema** pane, modify the `Query` type to add a new `allPostsByAuthor` query as follows:

```
type Query {
    allPostsByAuthor(author: String!, limit: Int, nextToken: String): PaginatedPosts!
    allPost(limit: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

   Note that this uses the same `PaginatedPosts` type that you used with the `allPost` query.
3. Choose **Save Schema**.
4. In the **Resolvers** pane on the right, find the newly created `allPostsByAuthor` field on the `Query` type, and then choose **Attach**. Create a resolver using the snippet below:

```
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 };
}
```

Like the `allPost` resolver, this resolver has two optional arguments:

- `limit` - Specifies the maximum number of items to return in a single call.
- `nextToken` - Retrieves the next set of results (the value for `nextToken` can be obtained from a previous call).

5. Save any changes made to your resolver.

For more information about the `Query` request, see the Query reference documentation.

Call the API to query all posts by author

Now that the resolver has been set up, AWS AppSync knows how to translate an incoming `allPostsByAuthor` mutation to a DynamoDB `Query` operation against the `author-index` index. You can now query the table to retrieve all the posts by a specific author.

Before this, however, let's populate the table with some more posts, because every post so far has the same author.

To add data and query

1. In your API, choose the **Queries** tab.
2. In the **Queries** pane, add the following mutation:

```
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. Choose **Run** (the orange play button), then choose `addPost`.
4. Now, let's query the table, returning all posts authored by `Nadia`. In the **Queries** pane, add the following query:

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

5. Choose **Run** (the orange play button), then choose `allPostsByAuthor`. All posts authored by `Nadia` should appear in the **Results** pane to the right of the **Queries** pane. It should look similar to the following:

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

6. Pagination works for `Query` just the same as it does for `Scan`. For example, let's look for all posts by `AUTHORNAME`, getting five at a time.

7. In the **Queries** pane, add the following query:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

8. Choose **Run** (the orange play button), then choose `allPostsByAuthor`. All posts authored by `AUTHORNAME` should appear in the **Results** pane to the right of the **Queries** pane. It should look similar to the following:

```
{
  "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. Update the `nextToken` argument with the value returned from the previous query as follows:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

10. Choose **Run** (the orange play button), then choose `allPostsByAuthor`. The remaining posts authored by `AUTHORNAME` should appear in the **Results** pane to the right of the **Queries** pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

Using sets

Up to this point, the `Post` type has been a flat key/value object. You can also model complex objects with your resolver, such as sets, lists, and maps. Let's update the `Post` type to include tags. A post can have zero or more tags, which are stored in DynamoDB as a String Set. You'll also set up some mutations to add and remove tags, and a new query to scan for posts with a specific tag.

To set up your data

1. In your API, choose the **Schema** tab.
2. In the **Schema** pane, modify the `Post` type to add a new `tags` field as follows:

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
}
```

3. In the **Schema** pane, modify the `Query` type to add a new `allPostsByTag` query as follows:

```
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. In the **Schema** pane, modify the `Mutation` type to add new `addTag` and `removeTag` mutations as follows:

```
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. Choose **Save Schema**.

6. In the **Resolvers** pane on the right, find the newly created `allPostsByTag` field on the `Query` type, and then choose **Attach**. Create your resolver using the snippet below:

```
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. Save any changes you've made to your resolver.
8. Now, do the same for the `Mutation` field `addTag` using the snippet below:

> **Note**
>
> Though the DynamoDB utils currently don't support set operations, you can still interact with sets by building the request yourself.

```
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. Save any changes made to your resolver.
10. Repeat this one more time for the `Mutation` field `removeTag` using the snippet below:

```
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
```

11Save any changes made to your resolver.

Call the API to work with tags

Now that you've set up the resolvers, AWS AppSync knows how to translate incoming `addTag`, `removeTag`, and `allPostsByTag` requests into DynamoDB `UpdateItem` and `Scan` operations. To try it out, let's select one of the posts you created earlier. For example, let's use a post authored by `Nadia`.

To use tags

1. In your API, choose the **Queries** tab.
2. In the **Queries** pane, add the following query:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

3. Choose **Run** (the orange play button), then choose `allPostsByAuthor`.
4. All of Nadia's posts should appear in the **Results** pane to the right of the **Queries** pane. It should look similar to the following:

```
{
  "data": {
```

```
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

5. Let's use the one with the title *The cutest dog in the world*. Record its `id` because you'll use it later. Now, let's try adding a dog tag.
6. In the **Queries** pane, add the following mutation. You'll also need to update the `id` argument to the value you noted down earlier.

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

7. Choose **Run** (the orange play button), then choose `addTag`. The post is updated with the new tag:

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
```

```
}
```

8. You can add more tags. Update the mutation to change the `tag` argument to `puppy`:

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

9. Choose **Run** (the orange play button), then choose `addTag`. The post is updated with the new tag:

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

10. You can also delete tags. In the **Queries** pane, add the following mutation. You'll also need to update the `id` argument to the value you noted down earlier:

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

11. Choose **Run** (the orange play button), then choose `removeTag`. The post is updated and the `puppy` tag is deleted.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

12. You can also search for all posts that have a tag. In the **Queries** pane, add the following query:

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

13. Choose **Run** (the orange play button), then choose `allPostsByTag`. All posts that have the dog tag are returned as follows:

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
            "dog",
            "puppy"
          ]
        }
      ],
      "nextToken": null
```

```
    }
  }
}
```

Conclusion

In this tutorial, you've built an API that lets you manipulate `Post` objects in DynamoDB using AWS AppSync and GraphQL.

To clean up, you can delete the AWS AppSync GraphQL API from the console.

To delete the role associated with your DynamoDB table, select your data source in the **Data Sources** table and click **edit**. Note the value of the role under **Create or use an existing role**. Go to the IAM console to delete the role.

To delete your DynamoDB table, click on the name of the table in the data sources list. This takes you to the DynamoDB console where you can delete the table.

Using AWS Lambda resolvers in AWS AppSync

You can use AWS Lambda with AWS AppSync to resolve any GraphQL field. For example, a GraphQL query might send a call to an Amazon Relational Database Service (Amazon RDS) instance, and a GraphQL mutation might write to an Amazon Kinesis stream. In this section, we'll show you how to write a Lambda function that performs business logic based on the invocation of a GraphQL field operation.

Create a Lambda function

The following example shows a Lambda function written in `Node.js` (runtime: Node.js 18.x) that performs different operations on blog posts as part of a blog post application. Note that the code should be saved in a file name with a .mis extension.

```
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)
  }
```

```
}
```

This Lambda function retrieves a post by ID, adds a post, retrieves a list of posts, and fetches related posts for a given post.

> **Note**
>
> The Lambda function uses the `switch` statement on `event.field` to determine which field is currently being resolved.

Create this Lambda function using the AWS Management Console.

Configure a data source for Lambda

After you create the Lambda function, navigate to your GraphQL API in the AWS AppSync console, and then choose the **Data Sources** tab.

Choose **Create data source**, enter a friendly **Data source name** (for example, **Lambda**), and then for **Data source type**, choose **AWS Lambda function**. For **Region**, choose the same Region as your function. For **Function ARN**, choose the Amazon Resource Name (ARN) of your Lambda function.

After choosing your Lambda function, you can either create a new AWS Identity and Access Management (IAM) role (for which AWS AppSync assigns the appropriate permissions) or choose an existing role that has the following inline policy:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:us-
east-1:111122223333:function:LAMBDA_FUNCTION"
        }
    ]
```

```
}
```

You must also set up a trust relationship with AWS AppSync for the IAM role as follows:

JSON

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

Create a GraphQL schema

Now that the data source is connected to your Lambda function, create a GraphQL schema.

From the schema editor in the AWS AppSync console, make sure that your schema matches the following schema:

```
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]
}
```

Configure resolvers

Now that you've registered a Lambda data source and a valid GraphQL schema, you can connect your GraphQL fields to your Lambda data source using resolvers.

You will create a resolver that uses the AWS AppSync JavaScript (APPSYNC_JS) runtime and interact with your Lambda functions. To learn more about writing AWS AppSync resolvers and functions with JavaScript, see JavaScript runtime features for resolvers and functions.

For more information about Lambda mapping templates, see JavaScript resolver function reference for Lambda.

In this step, you attach a resolver to the Lambda function for the following fields: `getPost(id:ID!): Post`, `allPosts: [Post]`, `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`, and `Post.relatedPosts: [Post]`. From the **Schema** editor in the AWS AppSync console, in the **Resolvers** pane, choose **Attach** next to the `getPost(id:ID!): Post` field. Choose your Lambda data source. Next, provide the following code:

```
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;
}
```

This resolver code passes the field name, list of arguments, and context about the source object to the Lambda function when it invokes it. Choose **Save**.

You have successfully attached your first resolver. Repeat this operation for the remaining fields.

Test your GraphQL API

Now that your Lambda function is connected to GraphQL resolvers, you can run some mutations and queries using the console or a client application.

On the left side of the AWS AppSync console, choose **Queries**, and then paste in the following code:

addPost Mutation

```
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 Query

```
query GetPost {
    getPost(id: "2") {
        id
        author
        title
        content
        url
```

```
        ups
        downs
    }
}
```

allPosts Query

```
query AllPosts {
    allPosts {
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {
            id
            title
        }
    }
}
```

Returning errors

Any given field resolution can result in an error. With AWS AppSync, you can raise errors from the following sources:

- Resolver response handler
- Lambda function

From the resolver response handler

To raise intentional errors, you can use the `util.error` utility method. It takes an argument an `errorMessage`, an `errorType`, and an optional `data` value. The `data` is useful for returning extra data back to the client when an error occurs. The `data` object is added to the `errors` in the GraphQL final response.

The following example shows how to use it in the `Post.relatedPosts: [Post]` resolver response handler.

```
// the Post.relatedPosts response handler
export function response(ctx) {
    util.error("Failed to fetch relatedPosts", "LambdaFailure", ctx.result)
    return ctx.result;
}
```

This yields a GraphQL response similar to the following:

```
{
    "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"
                }
```

```
            ]
        }
    ]
}
```

Where `allPosts[0].relatedPosts` is *null* because of the error and the `errorMessage`, `errorType`, and `data` are present in the `data.errors[0]` object.

From the Lambda function

AWS AppSync also understands errors that the Lambda function throws. The Lambda programming model lets you raise *handled* errors. If the Lambda function throws an error, AWS AppSync fails to resolve the current field. Only the error message returned from Lambda is set in the response. Currently, you can't pass any extraneous data back to the client by raising an error from the Lambda function.

> **Note**
>
> If your Lambda function raises an *unhandled* error, AWS AppSync uses the error message that Lambda set.

The following Lambda function raises an error:

```
export const handler = async (event) => {
  console.log('Received event {}', JSON.stringify(event, 3))
  throw new Error('I always fail.')
}
```

The error is received in your response handler. You can send it back in the GraphQL response by appending the error to the response with `util.appendError`. To do so, change your AWS AppSync function response handler to this:

```
// the lambdaInvoke response handler
export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  }
  return result;
```

```
}
```

This returns a GraphQL response similar to the following:

```
{
  "data": {
    "allPosts": null
  },
  "errors": [
    {
      "path": [
        "allPosts"
      ],
      "data": null,
      "errorType": "Lambda:Unhandled",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "I fail. always"
    }
  ]
}
```

Advanced use case: Batching

The Lambda function in this example has a `relatedPosts` field that returns a list of related posts for a given post. In the example queries, the `allPosts` field invocation from the Lambda function returns five posts. Because we specified that we also want to resolve `relatedPosts` for each returned post, the `relatedPosts` field operation is invoked five times.

```
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
        }
    }
}
```

While this might not sound substantial in this specific example, this compounded over-fetching can quickly undermine the application.

If you were to fetch `relatedPosts` again on the returned related `Posts` in the same query, the number of invocations would increase dramatically.

```
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
            }
        }
    }
}
```

In this relatively simple query, AWS AppSync would invoke the Lambda function 1 + 5 + 25 = 31 times.

This is a fairly common challenge and is often called the N+1 problem (in this case, N = 5), and it can incur increased latency and cost to the application.

One approach to solving this issue is to batch similar field resolver requests together. In this example, instead of having the Lambda function resolve a list of related posts for a single given post, it could instead resolve a list of related posts for a given batch of posts.

To demonstrate this, let's update the resolver for `relatedPosts` to handle batching.

```
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;
}
```

The code now changes the operation from `Invoke` to `BatchInvoke` when the `fieldName` being resolved is `relatedPosts`. Now, enable batching on the function in the **Configure Batching** section. Set the maximum batching size set to 5. Choose **Save**.

With this change, when resolving `relatedPosts`, the Lambda function receives the following as input:

```
[
    {
        "field": "relatedPosts",
        "source": {
            "id": 1
        }
    },
    {
        "field": "relatedPosts",
        "source": {
            "id": 2
```

```
        }
    },
    ...
]
```

When `BatchInvoke` is specified in the request, the Lambda function receives a list of requests and returns a list of results.

Specifically, the list of results must match the size and order of the request payload entries so that AWS AppSync can match the results accordingly.

In this batching example, the Lambda function returns a batch of results as follows:

```
[
    [{"id":"2","title":"Second book"}, {"id":"3","title":"Third book"}],   //
 relatedPosts for id=1
    [{"id":"3","title":"Third book"}]                                      //
 relatedPosts for id=2
]
```

You can update your Lambda code to handle batching for `relatedPosts`:

```
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)
  }
}
```

Returning individual errors

The previous examples show that it's possible to return a single error from the Lambda function or raise an error from your response handler. For batched invocations, raising an error from the Lambda function flags an entire batch as failed. This might be acceptable for specific scenarios

where an irrecoverable error occurs, such as a failed connection to a data store. However, in cases where some items in the batch succeed and others fail, it's possible to return both errors and valid data. Because AWS AppSync requires the batch response to list elements matching the original size of the batch, you must define a data structure that can differentiate valid data from an error.

For example, if the Lambda function is expected to return a batch of related posts, you could choose to return a list of `Response` objects where each object has optional *data*, *errorMessage*, and *errorType* fields. If the *errorMessage* field is present, it means that an error occurred.

The following code shows how you could update the Lambda function:

```
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)
  }
}
```

Update the `relatedPosts` resolver code:

```
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
  }
}
```

The response handler now checks for errors returned by the Lambda function on `Invoke` operations, checks for errors returned for individual items for `BatchInvoke` operations, and finally checks the `fieldName`. For `relatedPosts`, the function returns `result.data`. For all other fields, the function just returns `result`. For example, see the query below:

```
query AllPosts {
  allPosts {
    id
    title
    content
    url
    ups
    downs
    relatedPosts {
      id
    }
    author
  }
}
```

This query returns a GraphQL response similar to the following:

```
{
  "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"
    }
  ]
}
```

Configuring the maximum batching size

To configure the maximum batching size on a resolver, use the following command in the AWS Command Line Interface (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**
>
> When providing a request mapping template, you must use the `BatchInvoke` operation to use batching.

Using local resolvers in AWS AppSync

AWS AppSync allows you to use supported data sources (AWS Lambda, Amazon DynamoDB, or Amazon OpenSearch Service) to perform various operations. However, in certain scenarios, a call to a supported data source might not be necessary.

This is where the local resolver comes in handy. Instead of calling a remote data source, the local resolver will just **forward** the result of the request handler to the response handler. The field resolution will not leave AWS AppSync.

Local resolvers are useful in a plethora of situations. The most popular use case is to publish notifications without triggering a data source call. To demonstrate this use case, let's build a pub/sub application in which users can publish and subscribe to messages. This example leverages *Subscriptions*, so if you aren't familiar with *Subscriptions*, you can follow the Real-Time Data tutorial.

Creating the pub/sub app

First, create a blank GraphQL API by choosing the **Design from scratch** option and configuring the optional details when creating your GraphQL API.

In our pub/sub application, clients can subscribe to and publish messages. Each published message includes a name and data. Add this to the schema:

```
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"])
}
```

Next, let's attach a resolver to the `Mutation.publish` field. In the **Resolvers** pane next to the **Schema** pane, find the `Mutation` type, then the `publish(...): Channel` field, then click on **Attach**.

Create a *None* data source and name it *PageDataSource*. Attach it to your resolver.

Add your resolver implementation using the following snippet:

```
export function request(ctx) {
  return { payload: ctx.args };
}

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

Make sure you create the resolver and save the changes you made.

Send and subscribe to messages

For clients to receive messages, they must first be subscribed to an inbox.

In the **Queries** pane, execute the `SubscribeToData` subscription:

```
subscription SubscribeToData {
    subscribe(name:"channel") {
        name
        data
    }
}
```

The subscriber will receive messages whenever the `publish` mutation is invoked but only when the message is sent to the `channel` subscription. Let's try this in the **Queries** pane. While your subscription is still running in the console, open up another console and run the following request in the **Queries** pane:

> **Note**
>
> We're using valid JSON strings in this example.

```
mutation PublishData {
    publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
        data
        name
    }
```

```
}
```

The result will look like this:

```
{
  "data": {
    "publish": {
      "data": "{\"msg\":\"hello world!\"}",
      "name": "channel"
    }
  }
}
```

We just demonstrated the use of local resolvers, by publishing a message and receiving it without leaving the AWS AppSync service.

Combining GraphQL resolvers in AWS AppSync

Resolvers and fields in a GraphQL schema have 1:1 relationships with a large degree of flexibility. Because a data source is configured on a resolver independently of a schema, you have the ability to resolve or manipulate your GraphQL types through different data sources, allowing you to mix and match a schema to best meet your needs.

The following scenarios demonstrate how to mix and match data sources in your schema. Before you begin, you should be familiar with configuring data sources and resolvers for AWS Lambda, Amazon DynamoDB, and Amazon OpenSearch Service.

Example schema

The following schema has a type of `Post` with three `Query` and `Mutation` operations each:

```
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
}
```

In this example, you would have a total of six resolvers with each needing a data source. One way to solve this issue would be to hook these up to a single Amazon DynamoDB table, called `Posts`, in which the `AllPost` field runs a scan and the `searchPosts` field runs a query (see JavaScript resolver function reference for DynamoDB). However, you aren't limited to Amazon DynamoDB; different data sources like Lambda or OpenSearch Service exist to meet your business requirements.

Altering data through resolvers

You may need to return results from a third-party database that's not directly supported by AWS AppSync data sources. You may also have to perform complex modifications on the data before it's returned to the API client(s). This could be caused by the improper formatting of the data types, such as timestamp differences on clients, or the handling of backwards compatibility issues. In this case, connecting AWS Lambda functions as a data source to your AWS AppSync API is the

appropriate solution. For illustrative purposes, in the following example, an AWS Lambda function manipulates data fetched from a third-party data store:

```
export const handler = (event, context, callback) => {
    // fetch data
    const result = fetcher()

    // apply complex business logic
    const data = transform(result)

    // return to AppSync
    return data
};
```

This is a perfectly valid Lambda function and could be attached to the `AllPost` field in the GraphQL schema so that any query returning all the results gets random numbers for the ups/downs.

DynamoDB and OpenSearch Service

For some applications, you might perform mutations or simple lookup queries against DynamoDB and have a background process transfer documents to OpenSearch Service. You could simply attach the `searchPosts` resolver to the OpenSearch Service data source and return search results (from data that originated in DynamoDB) using a GraphQL query. This can be extremely powerful when adding advanced search operations to your applications such keyword, fuzzy word matches, or even geospatial lookups. Transferring data from DynamoDB could be done through an ETL process, or alternatively, you could stream from DynamoDB using Lambda.

To get started with these particular data sources, see our DynamoDB and Lambda tutorials.

For example, using the schema from our previous tutorial, the following mutation adds an item to DynamoDB:

```
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
  }
}
```

This writes data to DynamoDB, which then streams data via Lambda to Amazon OpenSearch Service, which you then use to search for posts by different fields. For example, since the data is in Amazon OpenSearch Service, you can search either the author or content fields with free-form text, even with spaces, as follows:

```
query searchName{
    searchAuthor(name:"   Nadia   "){
        id
        title
        content
    }
}

---------- or ----------

query searchContent{
    searchContent(text:"test"){
        id
        title
        content
    }
}
```

Because the data is written directly to DynamoDB, you can still perform efficient list or item lookup operations against the table with the `allPost{...}` and `getPost{...}` queries. This stack uses the following example code for DynamoDB streams:

> **Note**
>
> This Python code is an example and isn't meant to be used in production code.

```
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.'
```

You can then use DynamoDB streams to attach this to a DynamoDB table with a primary key of `id`, and any changes to the source of DynamoDB would stream into your OpenSearch Service domain. For more information about configuring this, see the DynamoDB Streams documentation.

Using Amazon OpenSearch Service resolvers in AWS AppSync

AWS AppSync supports using Amazon OpenSearch Service from domains that you have provisioned in your own AWS account, provided they don't exist inside a VPC. After your domains are provisioned, you can connect to them using a data source, at which point you can configure

a resolver in the schema to perform GraphQL operations such as queries, mutations, and subscriptions. This tutorial will take you through some common examples.

For more information, see our JavaScript resolver function reference for OpenSearch.

Create a new OpenSearch Service domain

To get started with this tutorial, you need an existing OpenSearch Service domain. If you don't have one, you can use the following sample. Note that it can take up to 15 minutes for an OpenSearch Service domain to be created before you can move on to integrating it with an AWS AppSync data source.

```
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
```

You can launch the following AWS CloudFormation stack in the US-West-2 (Oregon) Region in your AWS account:

Configure a data source for OpenSearch Service

After the OpenSearch Service domain is created, navigate to your AWS AppSync GraphQL API and choose the **Data Sources** tab. Choose **Create data source** and enter a friendly name for the data source such as "*oss*". Then, choose **Amazon OpenSearch domain** for **Data source type**, choose the appropriate Region, and you should see your OpenSearch Service domain listed. After selecting it, you can either create a new role, and AWS AppSync will assign the role-appropriate permissions, or you can choose an existing role, which has the following inline policy:

You'll also need to set up a trust relationship with AWS AppSync for that role:

JSON

```
{
    "Version": "2012-10-17",
```

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

Additionally, the OpenSearch Service domain has its own **Access Policy** that you can modify through the Amazon OpenSearch Service console. You must add a policy similar to the one below with the appropriate actions and resources for the OpenSearch Service domain. Note that the **Principal** will be the AWS AppSync data source role, which can be found in the IAM console if you let said console create it.

Connecting a resolver

Now that the data source is connected to your OpenSearch Service domain, you can connect it to your GraphQL schema with a resolver as shown in the following example:

```
 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
}
```

Note that there is a user-defined `Post` type with a field of `id`. In the following examples, we assume there is a process (which can be automated) for putting this type into your OpenSearch Service domain, which would map to a path root of `/post/_doc` where `post` is the index. From this root path, you can perform individual document searches, wildcard searches with `/id/post*`, or multi-document searches with a path of `/post/_search`. For example, if you have another type called `User`, you can index documents under a new index called `user`, then perform searches with a **path** of `/user/_search`.

From the **Schema** editor in the AWS AppSync console, modify the preceding `Posts` schema to include a `searchPosts` query:

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

Save the schema. In the **Resolvers** pane, find `searchPosts` and choose **Attach**. Choose your OpenSearch Service data source and save the resolver. Update your resolver's code using the snippet below:

```
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)
}
```

This assumes that the preceding schema has documents that have been indexed in OpenSearch Service under the `post` field. If you structure your data differently, you'll need to update accordingly.

Modifying your searches

The preceding resolver request handler performs a simple query for all records. Suppose you want to search by a specific author. Furthermore, suppose you want that author to be an argument defined in your GraphQL query. In the **Schema** editor of the AWS AppSync console, add an `allPostsByAuthor` query:

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

In the **Resolvers** pane, find `allPostsByAuthor` and choose **Attach**. Choose the OpenSearch Service data source and use the following code:

```
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)
}
```

Note that the body is populated with a term query for the `author` field, which is passed through from the client as an argument. Optionally, you could use prepopulated information, such as standard text.

Adding data to OpenSearch Service

You may want to add data to your OpenSearch Service domain as the result of a GraphQL mutation. This is a powerful mechanism for searching and other purposes. Because you can use GraphQL subscriptions to make your data real-time, it can serve as a mechanism for notifying clients of updates to data in your OpenSearch Service domain.

Return to the **Schema** page in the AWS AppSync console and select **Attach** for the `addPost()` mutation. Select the OpenSearch Service data source again and use the following code:

```
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
}
```

Like before, this is an example of how your data might be structured. If you have different field names or indices, you need to update the `path` and `body`. This example also shows how to use `context.arguments`, which can also be written as `ctx.args`, in your request handler.

Retrieving a single document

Finally, if you want to use the `getPost(id:ID)` query in your schema to return an individual document, find this query in the **Schema** editor of the AWS AppSync console and choose **Attach**. Select the OpenSearch Service data source again and use the following code:

```
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
}
```

Perform queries and mutations

You should now be able to perform GraphQL operations against your OpenSearch Service domain. Navigate to the **Queries** tab of the AWS AppSync console and add a new record:

```
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
       )
}
```

You'll see the result of the mutation on the right. Similarly, you can now run a `searchPosts` query against your OpenSearch Service domain:

```
query search {
    searchPosts {
        id
        title
        author
        content
    }
}
```

Best practices

- OpenSearch Service should be for querying data, not as your primary database. You may want to use OpenSearch Service in conjunction with Amazon DynamoDB as outlined in Combining GraphQL Resolvers.
- Only give access to your domain by allowing the AWS AppSync service role to access the cluster.
- You can start small in development, with the lowest-cost cluster, and then move to a larger cluster with high availability (HA) as you move into production.

Performing DynamoDB transactions in AWS AppSync

AWS AppSync supports using Amazon DynamoDB transaction operations across one or more tables in a single Region. Supported operations are `TransactGetItems` and `TransactWriteItems`. By using these features in AWS AppSync, you can perform tasks such as:

- Passing a list of keys in a single query and returning the results from a table
- Reading records from one or more tables in a single query
- Writing records in transactions to one or more tables in an all-or-nothing way
- Running transactions when some conditions are satisfied

Permissions

Like other resolvers, you need to create a data source in AWS AppSync and either create a role or use an existing one. Because transaction operations require different permissions on DynamoDB tables, you need to grant the configured role permissions for read or write actions:

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-1:111122223333:table/TABLENAME",
                "arn:aws:dynamodb:us-east-1:111122223333:table/TABLENAME/*"
            ]
        }
    ]
}
```

Note

Roles are tied to data sources in AWS AppSync, and resolvers on fields are invoked against a data source. Data sources configured to fetch against DynamoDB only have one table specified to keep configurations simple. Therefore, when performing a transaction operation against multiple tables in a single resolver, which is a more advanced task, you must grant the role on that data source access to any tables the resolver will interact with. This would be done in the **Resource** field in the IAM policy above. Configuration of the transaction calls against the tables is done in the resolver code, which we describe below.

Data source

For the sake of simplicity, we'll use the same data source for all the resolvers used in this tutorial.

We'll have two tables called **savingAccounts** and **checkingAccounts**, both with the `accountNumber` as a partition key, and a **transactionHistory** table with `transactionId` as partition key. You can use the CLI commands below to create your tables. Make sure to replace `region` with your Region.

With the CLI

```
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
```

In the AWS AppSync console, in **Data sources**, create a new DynamoDB data source and name it **TransactTutorial**. Select **savingAccounts** as the table (though the specific table does not matter when using transactions). Choose to create a new role and the data source. You can review the data source configuration to see the name of the generated role. In the IAM console, you can add an in-line policy that allows the data source to interact with all the tables.

Replace `region` and `accountID` with your Region and account ID:

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-1:111122223333:table/savingAccounts",
                "arn:aws:dynamodb:us-east-1:111122223333:table/savingAccounts/*",
                "arn:aws:dynamodb:us-east-1:111122223333:table/checkingAccounts",
                "arn:aws:dynamodb:us-east-1:111122223333:table/checkingAccounts/
*",
```

```
                "arn:aws:dynamodb:us-east-1:111122223333:table/
transactionHistory",
                "arn:aws:dynamodb:us-east-1:111122223333:table/
transactionHistory/*"
            ]
        }
    ]
}
```

Transactions

For this example, the context is a classic banking transaction, where we'll use `TransactWriteItems` to:

- Transfer money from saving accounts to checking accounts
- Generate new transaction records for each transaction

And then we'll use `TransactGetItems` to retrieve details from saving accounts and checking accounts.

> **Warning**
>
> `TransactWriteItems` is not supported when used with conflict detection and resolution. These settings must be disabled to prevent possible errors.

We define our GraphQL schema as follows:

```
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 - Populate accounts

In order to transfer money between accounts, we need to populate the table with the details. We'll use the GraphQL operation `Mutation.populateAccounts` to do so.

In the Schema section, click on **Attach** next to the `Mutation.populateAccounts` operation. Choose the `TransactTutorial` data source and choose **Create**.

Now use the following code:

```
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 }
}
```

Save the resolver and navigate to the **Queries** section of the AWS AppSync console to populate the accounts.

Execute the following mutation:

```
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
    }
  }
}
```

We populated three saving accounts and three checking accounts in one mutation.

Use the DynamoDB console to validate that data shows up in both the **savingAccounts** and **checkingAccounts** tables.

TransactWriteItems - Transfer money

Attach a resolver to the `transferMoney` mutation with the following code. For each transfer, we need a success modifier to both the checking and savings accounts, and we need to track the transfer in transactions.

```
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 }
}
```

Now, navigate to the **Queries** section of the AWS AppSync console and execute the **transferMoney** mutation as follows:

```
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
    }
  }
```

```
}
```

We sent three banking transactions in one mutation. Use the DynamoDB console to validate that data shows up in the **savingAccounts**, **checkingAccounts**, and **transactionHistory** tables.

TransactGetItems - Retrieve accounts

In order to retrieve the details from savings and checking accounts in a single transactional request, we'll attach a resolver to the `Query.getAccounts` GraphQL operation on our schema. Select **Attach**, pick the same `TransactTutorial` data source created at the beginning of the tutorial. Use the following code:

```
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 }
```

```
}
```

Save the resolver and navigate to the **Queries** sections of the AWS AppSync console. In order to retrieve the savings and checking accounts, execute the following query:

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
      username
      balance
    }
  }
}
```

We have successfully demonstrated the use of DynamoDB transactions using AWS AppSync.

Using DynamoDB batch operations in AWS AppSync

AWS AppSync supports using Amazon DynamoDB batch operations across one or more tables in a single Region. Supported operations are `BatchGetItem`, `BatchPutItem`, and `BatchDeleteItem`. By using these features in AWS AppSync, you can perform tasks such as:

- Passing a list of keys in a single query and returning the results from a table
- Reading records from one or more tables in a single query
- Writing records in bulk to one or more tables
- Conditionally writing or deleting records in multiple tables that might have a relation

Batch operations in AWS AppSync have two key differences from non-batched operations:

- The data source role must have permissions to all tables that the resolver will access.

- The table specification for a resolver is part of the request object.

Single table batches

> **Warning**
>
> `BatchPutItem` and `BatchDeleteItem` are not supported when used with conflict detection and resolution. These settings must be disabled to prevent possible errors.

To get started, let's create a new GraphQL API. In the AWS AppSync console, choose **Create API**, **GraphQL APIs**, and **Design from scratch**. Name your API `BatchTutorial API`, choose **Next**, and on the **Specify GraphQL resources** step, choose **Create GraphQL resources later** and click **Next**. Review your details and create the API. Go to the **Schema** page and paste the following schema, noting that for the query, we'll pass in a list of IDs:

```
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]
}
```

Save your schema and choose **Create Resources** at the top of the page. Choose **Use existing type** and select the `Post` type. Name your table `Posts`. Make sure the **Primary Key** is set to `id`, unselect **Automatically generate GraphQL** (you'll provide your own code), and select **Create**. To get you started, AWS AppSync creates a new DynamoDB table and a data source connected to the

table with the appropriate roles. However, there are still a couple of permissions you need to add to the role. Go to the **Data sources** page and choose the new data source. Under **Select an existing role**, you'll notice that a role was automatically created for the table. Take note of the role (should look something like `appsync-ds-ddb-aaabbbcccddd-Posts`) and then go to the IAM console (https://console.aws.amazon.com/iam/). In the IAM console, choose **Roles**, then choose your role from the table. In your role, under **Permissions policies**, click on the "+" next to the policy (should have a similar name to the role name). Choose **Edit** at the top of the collapsible when the policy appears. You need to add batch permissions to your policy, specifically `dynamodb:BatchGetItem` and `dynamodb:BatchWriteItem`. It'll look something like this:

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:us-east-1:111122223333:table/locationReadings",
                "arn:aws:dynamodb:us-east-1:111122223333:table/locationReadings/
*",
                "arn:aws:dynamodb:us-east-1:111122223333:table/
temperatureReadings",
                "arn:aws:dynamodb:us-east-1:111122223333:table/
temperatureReadings/*"
            ]
        }
    ]
}
```

Choose **Next**, then **Save changes**. Your policy should allow batch processing now.

Back in the AWS AppSync console, go to the **Schema** page and select **Attach** next to the `Mutation.batchAdd` field. Create your resolver using the `Posts` table as the data source. In the code editor, replace the handlers with the snippet below. This snippet automatically takes each item in the GraphQL `input PostInput` type and builds a map, which is needed for the `BatchPutItem` operation:

```
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;
}
```

Navigate to the **Queries** page of the AWS AppSync console and run the following `batchAdd` mutation:

```
mutation add {
    batchAdd(posts:[{
            id: 1 title: "Running in the Park"},{
            id: 2 title: "Playing fetch"
        }]){
            id
            title
    }
}
```

You should see the results printed on the screen; this can be validated by reviewing the DynamoDB console to scan for the values written to the `Posts` table.

Next, repeat the process of attaching a resolver but for the `Query.batchGet` field using the `Posts` table as the data source. Replace the handlers with the code below. This automatically takes each item in the GraphQL `ids:[]` type and builds a map that is needed for the `BatchGetItem` operation:

```
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;
}
```

Now, go back to the **Queries** page of the AWS AppSync console and run the following `batchGet` query:

```
query get {
    batchGet(ids:[1,2,3]){
        id
        title
    }
}
```

This should return the results for the two `id` values that you added earlier. Note that a `null` value was returned for the `id` with a value of `3`. This is because there was no record in your `Posts` table with that value yet. Also note that AWS AppSync returns the results in the same order as the keys passed to the query, which is an additional feature that AWS AppSync performs on your behalf. So,

if you switch to `batchGet(ids:[1,3,2])`, you'll see that the order changed. You'll also know which `id` returned a `null` value.

Finally, attach one more resolver to the `Mutation.batchDelete` field using the `Posts` table as the data source. Replace the handlers with the code below. This automatically takes each item in the GraphQL `ids:[]` type and builds a map that is needed for the `BatchGetItem` operation:

```
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;
}
```

Now, go back to the **Queries** page of the AWS AppSync console and run the following `batchDelete` mutation:

```
mutation delete {
    batchDelete(ids:[1,2]){ id }
}
```

The records with `id` 1 and 2 should now be deleted. If you re-run the `batchGet()` query from earlier, these should return `null`.

Multi-table batch

> **Warning**
>
> `BatchPutItem` and `BatchDeleteItem` are not supported when used with conflict detection and resolution. These settings must be disabled to prevent possible errors.

AWS AppSync also enables you to perform batch operations across tables. Let's build a more complex application. Imagine we are building a pet health app wherein sensors report the pet's location and body temperature. The sensors are battery powered and attempt to connect to the network every few minutes. When a sensor establishes a connection, it sends its readings to our AWS AppSync API. Triggers then analyze the data so a dashboard can be presented to the pet owner. Let's focus on representing the interactions between the sensor and the backend data store.

In the AWS AppSync console, choose **Create API**, **GraphQL APIs**, and **Design from scratch**. Name your API `MultiBatchTutorial API`, choose **Next**, and on the **Specify GraphQL resources** step, choose **Create GraphQL resources later** and click **Next**. Review your details and create the API. Go to the **Schema** page and paste and save the following schema:

```
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
}
```

We need to create two DynamoDB tables:

- `locationReadings` will store sensor location readings.
- `temperatureReadings` will store sensor temperature readings.

Both tables will share the same primary key structure: `sensorId (String)` as the partition key and `timestamp (String)` as the sort key.

Choose **Create Resources** at the top of the page. Choose **Use existing type** and select the `locationReadings` type. Name your table `locationReadings`. Make sure the **Primary Key** is set to `sensorId` and the sort key to `timestamp`. Unselect **Automatically generate GraphQL**

(you'll provide your own code), and select **Create**. Repeat this process for `temperatureReadings` using the `temperatureReadings` as the type and table name. Use the same keys as above.

Your new tables will contain automatically generated roles. There are still a couple of permissions you need to add to those roles. Go to the **Data sources** page and choose `locationReadings`. Under **Select an existing role**, you can see the role. Take note of the role (should look something like `appsync-ds-ddb-aaabbbcccddd-locationReadings`) and then go to the IAM console (https://console.aws.amazon.com/iam/). In the IAM console, choose **Roles**, then choose your role from the table. In your role, under **Permissions policies**, click on the "+" next to the policy (should have a similar name to the role name). Choose **Edit** at the top of the collapsible when the policy appears. You need to add permissions to this policy. It'll look something like this:

Choose **Next**, then **Save changes**. Repeat this process for the `temperatureReadings` data source using the same policy snippet above.

BatchPutItem - Recording sensor readings

Our sensors need to be able to send their readings once they connect to the internet. The GraphQL field `Mutation.recordReadings` is the API they will use to do so. We'll need to add a resolver to this field.

In the AWS AppSync console's **Schema** page, select **Attach** next to the `Mutation.recordReadings` field. On the next screen, create your resolver using the `locationReadings` table as the data source.

After creating your resolver, replace the handlers with the following code in the editor. This `BatchPutItem` operation allows us to specify multiple tables:

```
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
}
```

With batch operations, there can be both errors and results returned from the invocation. In that case, we're free to do some extra error handling.

> **Note**
>
> The use of `utils.appendError()` is similar to the `util.error()`, with the major distinction that it doesn't interrupt the evaluation of the request or response handler. Instead, it signals there was an error with the field but allows the handler to be evaluated and consequently return data back to the caller. We recommend that you use `utils.appendError()` when your application needs to return partial results.

Save the resolver and navigate to the **Queries** page in the AWS AppSync console. We can now send some sensor readings.

Execute the following mutation:

```
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
    }
  }
}
```

We sent ten sensor readings in one mutation with readings split up across two tables. Use the DynamoDB console to validate that the data shows up in both the `locationReadings` and `temperatureReadings` tables.

BatchDeleteItem - Deleting sensor readings

Similarly, we would also need to be able to delete batches of sensor readings. Let's use the `Mutation.deleteReadings` GraphQL field for this purpose. In the AWS AppSync console's **Schema** page, select **Attach** next to the `Mutation.deleteReadings` field. On the next screen, create your resolver using the `locationReadings` table as the data source.

After creating your resolver, replace the handlers in the code editor with the snippet below. In this resolver, we use a helper function mapper that extracts the `sensorId` and the `timestamp` from the provided inputs.

```
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
}
```

Save the resolver and navigate to the **Queries** page in the AWS AppSync console. Now, let's delete a couple of sensor readings.

Execute the following mutation:

```
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**
>
> Contrary to the `DeleteItem` operation, the fully deleted item isn't returned in the response. Only the passed key is returned. To learn more, see the BatchDeleteItem in JavaScript resolver function reference for DynamoDB .

Validate through the DynamoDB console that these two readings have been deleted from the `locationReadings` and `temperatureReadings` tables.

BatchGetItem - Retrieve readings

Another common operation for our app would be to retrieve the readings for a sensor at a specific point in time. Let's attach a resolver to the `Query.getReadings` GraphQL field on our schema. In the AWS AppSync console's **Schema** page, select **Attach** next to the `Query.getReadings` field. On the next screen, create your resolver using the `locationReadings` table as the data source.

Let's use the following code:

```
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' })),
 ]
}
```

Save the resolver and navigate to the **Queries** page in the AWS AppSync console. Now, let's retrieve our sensor readings.

Execute the following query:

```
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
    }
  }
}
```

We have successfully demonstrated the use of DynamoDB batch operations using AWS AppSync.

Error handling

In AWS AppSync, data source operations can sometimes return partial results. Partial results is the term we will use to denote when the output of an operation is comprised of some data and an error. Because error handling is inherently application specific, AWS AppSync gives you the opportunity to handle errors in the response handler. The resolver invocation error, if present, is available from the context as `ctx.error`. Invocation errors always include a message and a type, accessible as properties `ctx.error.message` and `ctx.error.type`. In the response handler, you can handle partial results in three ways:

1. Swallow the invocation error by just returning data.
2. Raise an error (using `util.error(...)`) by stopping the handler evaluation, which won't return any data.
3. Append an error (using `util.appendError(...)`) and also return data.

Let's demonstrate each of the three points above with DynamoDB batch operations.

DynamoDB Batch operations

With DynamoDB batch operations, it is possible that a batch partially completes. That is, it is possible that some of the requested items or keys are left unprocessed. If AWS AppSync is unable to complete a batch, unprocessed items and an invocation error will be set on the context.

We will implement error handling using the `Query.getReadings` field configuration from the `BatchGetItem` operation from the previous section of this tutorial.
This time, let's pretend that while executing the `Query.getReadings` field, the `temperatureReadings` DynamoDB table ran out of provisioned throughput. DynamoDB raised a `ProvisionedThroughputExceededException` during the second attempt by AWS AppSync to process the remaining elements in the batch.

The following JSON represents the serialized context after the DynamoDB batch invocation but before the response handler was called:

```
{
  "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": []
}
```

A few things to note on the context:

- The invocation error has been set on the context at `ctx.error` by AWS AppSync, and the error type has been set to `DynamoDB:ProvisionedThroughputExceededException`.
- Results are mapped per table under `ctx.result.data` even though an error is present.
- Keys that were left unprocessed are available at `ctx.result.data.unprocessedKeys`. Here, AWS AppSync was unable to retrieve the item with key (sensorId:1, timestamp:2018-02-01T17:21:05.000+08:00) because of insufficient table throughput.

> **Note**
>
> For `BatchPutItem`, it is `ctx.result.data.unprocessedItems`. For `BatchDeleteItem`, it is `ctx.result.data.unprocessedKeys`.

Let's handle this error in three different ways.

1. Swallowing the invocation error

Returning data without handling the invocation error effectively swallows the error, making the result for the given GraphQL field always successful.

The code we write is familiar and only focuses on the result data.

Response handler

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

GraphQL response

```
{
  "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
      }
    ]
  }
}
```

No errors will be added to the error response as only data was acted on.

2. Raising an error to abort the response handler execution

When partial failures should be treated as complete failures from the client's perspective, you can abort the response handler execution to prevent returning data. The `util.error(...)` utility method achieves exactly this behavior.

Response handler code

```
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 response

```
{
  "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. (...)"
    }
  ]
}
```

Even though some results might have been returned from the DynamoDB batch operation, we chose to raise an error such that the `getReadings` GraphQL field is null and the error has been added to the GraphQL response *errors* block.

3. Appending an error to return both data and errors

In certain cases, to provide a better user experience, applications can return partial results and notify their clients of the unprocessed items. The clients can decide to either implement a retry or translate the error back to the end user. The `util.appendError(...)` is the utility method

that enables this behavior by letting the application designer append errors on the context without interfering with the evaluation of the response handler. After evaluating the response handler, AWS AppSync will process any context errors by appending them to the errors block of the GraphQL response.

Response handler code

```
export function response(ctx) {
  if (ctx.error) {
    util.appendError(ctx.error.message, ctx.error.type, null,
 ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}
```

We forwarded both the invocation error and `unprocessedKeys` element inside the errors block of the GraphQL response. The `getReadings` field also return partial data from the `locationReadings` table as you can see in the response below.

GraphQL response

```
{
  "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. (...)"
    }
  ]
}
```

Using HTTP resolvers in AWS AppSync

AWS AppSync enables you to use supported data sources (that is, AWS Lambda, Amazon DynamoDB, Amazon OpenSearch Service, or Amazon Aurora) to perform various operations, in addition to any arbitrary HTTP endpoints to resolve GraphQL fields. After your HTTP endpoints are available, you can connect to them using a data source. Then, you can configure a resolver in the schema to perform GraphQL operations such as queries, mutations, and subscriptions. This tutorial walks you through some common examples.

In this tutorial you use a REST API (created using Amazon API Gateway and Lambda) with an AWS AppSync GraphQL endpoint.

Creating a REST API

You can use the following AWS CloudFormation template to set up a REST endpoint that works for this tutorial:

The AWS CloudFormation stack performs the following steps:

1. Sets up a Lambda function that contains your business logic for your microservice.

2. Sets up an API Gateway REST API with the following endpoint/method/content type combination:

API Resource Path	HTTP Method	Supported Content Type
/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

Creating your GraphQL API

To create the GraphQL API in AWS AppSync:

1. Open the AWS AppSync console and choose **Create API**.
2. Choose **GraphQL APIs** and then choose **Design from scratch**. Choose **Next**.
3. For the API name, type `UserData`. Choose **Next**.
4. Choose `Create GraphQL resources later`. Choose **Next**.
5. Review your inputs and choose **Create API**.

The AWS AppSync console creates a new GraphQL API for you using the API key authentication mode. You can use the console to further configure your GraphQL API and run requests.

Creating a GraphQL schema

Now that you have a GraphQL API, let's create a GraphQL schema. In the **Schema** editor in the AWS AppSync console, use the snippet below:

```
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
}
```

Configure your HTTP data source

To configure your HTTP data source, do the following:

1. In the **Data sources** page in your AWS AppSync GraphQL API, choose **Create data source**.
2. Enter a name for the data source like `HTTP_Example`.
3. In **Data source type**, choose **HTTP endpoint**.
4. Set the endpoint to the API Gateway endpoint that was created at the beginning of the tutorial. You can find your stack-generated endpoint if you navigate to the Lambda console and find your application under **Applications**. Inside of your application's settings, you should see an **API endpoint** which will be your endpoint in AWS AppSync. Make sure you don't include the stage name as part of the endpoint. For instance, if your endpoint were `https://aaabbbcccd.execute-api.us-east-1.amazonaws.com/v1`, you would type in `https://aaabbbcccd.execute-api.us-east-1.amazonaws.com`.

> **Note**
>
> At this time, only public endpoints are supported by AWS AppSync.
> For more information about the certifying authorities that are recognized by the AWS AppSync service, see Certificate Authorities (CA) Recognized by AWS AppSync for HTTPS Endpoints.

Configuring resolvers

In this step, you will connect the HTTP data source to the `getUser` and `addUser` queries.

To set up the `getUser` resolver:

1. In your AWS AppSync GraphQL API, choose the **Schema** tab.
2. To the right of the **Schema** editor, in the **Resolvers** pane and under the **Query** type, find the `getUser` field and choose **Attach**.
3. Keep the resolver type to `Unit` and the runtime to `APPSYNC_JS`.
4. In **Data source name**, choose the HTTP endpoint you made earlier.
5. Choose **Create**.
6. In the **Resolver** code editor, add the following snippet as your request handler:

```
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. Add the following snippet as your response handler:

```
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. Choose the **Query** tab, and then run the following query:

```
query GetUser{
    getUser(id:1){
        id
        username
    }
}
```

This should return the following response:

```
{
    "data": {
        "getUser": {
            "id": "1",
            "username": "nadia"
        }
    }
}
```

To set up the `addUser` resolver:

1. Choose the **Schema** tab.
2. To the right of the **Schema** editor, in the **Resolvers** pane and under the **Query** type, find the `addUser` field and choose **Attach**.
3. Keep the resolver type to `Unit` and the runtime to `APPSYNC_JS`.
4. In **Data source name**, choose the HTTP endpoint you made earlier.
5. Choose **Create**.

6. In the **Resolver** code editor, add the following snippet as your request handler:

```
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. Add the following snippet as your response handler:

```
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. Choose the **Query** tab, and then run the following query:

```
mutation addUser{
    addUser(userInput:{
        id:"2",
        username:"shaggy"
    }){
        id
        username
    }
}
```

If you run the `getUser` query again, it should return the following response:

```
{
    "data": {
        "getUser": {
        "id": "2",
        "username": "shaggy"
        }
    }
}
```

Invoking AWS Services

You can use HTTP resolvers to set up a GraphQL API interface for AWS services. HTTP requests to AWS must be signed with the Signature Version 4 process so that AWS can identify who sent them. AWS AppSync calculates the signature on your behalf when you associate an IAM role with the HTTP data source.

You provide two additional components to invoke AWS services with HTTP resolvers:

- An IAM role with permissions to call the AWS service APIs
- Signing configuration in the data source

For example, if you want to call the ListGraphqlApis operation with HTTP resolvers, you first create an IAM role that AWS AppSync assumes with the following policy attached:

JSON

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

Next, create the HTTP data source for AWS AppSync. In this example, you call AWS AppSync in the US West (Oregon) Region. Set up the following HTTP configuration in a file named `http.json`, which includes the signing region and service name:

```
{
    "endpoint": "https://appsync.us-west-2.amazonaws.com/",
    "authorizationConfig": {
        "authorizationType": "AWS_IAM",
        "awsIamConfig": {
            "signingRegion": "us-west-2",
            "signingServiceName": "appsync"
        }
    }
}
```

Then, use the AWS CLI to create the data source with an associated role as follows:

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

When you attach a resolver to the field in the schema, use the following request mapping template to call AWS AppSync:

```
{
    "version": "2018-05-29",
    "method": "GET",
    "resourcePath": "/v1/apis"
}
```

When you run a GraphQL query for this data source, AWS AppSync signs the request using the role you provided and includes the signature in the request. The query returns a list of AWS AppSync GraphQL APIs in your account in that AWS Region.

Using Aurora PostgreSQL with Data API in AWS AppSync

Learn how to connect your GraphQL API to Aurora PostgreSQL databases using AWS AppSync. This integration enables you to build scalable, data-driven applications by executing SQL queries and

mutations through GraphQL operations. AWS AppSync provides a data source for executing SQL statements against Amazon Aurora clusters that are enabled with a Data API. You can use AWS AppSync resolvers to run SQL statements against the data API with GraphQL queries, mutations, and subscriptions.

Before starting this tutorial, you should have basic familiarity with AWS services and GraphQL concepts.

> **Note**
>
> This tutorial uses the US-EAST-1 Region.

Topics

Set up your Aurora PostgreSQL database

Before adding an Amazon RDS data source to AWS AppSync, do the following.

1. Enable a Data API on an Aurora Serverless v2 cluster.
2. Configure a secret using AWS Secrets Manager
3. Create the cluster using the following AWS CLI command.

```
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
```

This will return an ARN for the cluster. After creating a cluster, you must add a Serverless v2 instance with the following AWS CLI command.

```
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**
>
> These endpoints take time to become activate. You can check their status in the RDS console in the **Connectivity & security** tab for the cluster.

Check the cluster status with the following AWS CLI command.

```
aws rds describe-db-clusters \
    --db-cluster-identifier appsync-tutorial \
    --query "DBClusters[0].Status"
```

Create a Secret via the AWS Secrets Manager Console or the AWS CLI with an input file such as the following using the `USERNAME` and `COMPLEX_PASSWORD` from the previous step:

```
{
    "username": "USERNAME",
    "password": "COMPLEX_PASSWORD"
}
```

Pass this as a parameter to the AWS CLI:

```
aws secretsmanager create-secret \
    --name appsync-tutorial-rds-secret \
    --secret-string file://creds.json
```

This will return an ARN for the secret. **Take note** of the ARN of your Aurora Serverless v2 cluster and Secret for later when creating a data source in the AWS AppSync console.

Creating the database and table

First, create a database named TESTDB. In PostgreSQL, a database is a container that holds tables and other SQL objects. Validate that your Aurora Serverless v2 cluster is configured correctly before adding it to your AWS AppSync API. First, create a *TESTDB* database with the `--sql` parameter as follows.

```
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"
```

If this runs without any errors, add two tables with the `create table` command:

```
 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);'
```

If successful, add the cluster as a data source in your API.

Creating a GraphQL schema

Now that your Aurora Serverless v2 Data API is running with configured tables, we'll create a GraphQL schema. You can quickly create your API by importing table configurations from an existing database using the API creation wizard.

To begin:

1. In the AWS AppSync console, choose **Create API**, then **Start with an Amazon Aurora cluster**.
2. Specify API details like **API name**, then select your database to generate the API.
3. Choose your database. If needed, update the Region, then choose your Aurora cluster and *TESTDB* database.
4. Choose your Secret, then choose **Import**.
5. Once tables have been discovered, update the type names. Change `Todos` to `Todo` and `Tasks` to `Task`.
6. Preview the generated schema by choosing **Preview Schema**. Your schema will look something like this:

```
type Todo {
  id: Int!
  description: String!
  due: AWSDate!
  createdAt: String
}

type Task {
  id: Int!
  todoId: Int!
  description: String
}
```

7. For the role, you can either have AWS AppSync create a new role or create one with a policy similar to the one below:

   JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "rds-data:ExecuteStatement"
            ],
            "Resource": [
                "arn:aws:rds:us-east-1:111122223333:cluster:appsync-tutorial",
                "arn:aws:rds:us-east-1:111122223333:cluster:appsync-tutorial:*"
            ]
        },
```

```
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": [
            "arn:aws:secretsmanager:us-east-1:111122223333:secret:appsync-
tutorial-rds-secret",
            "arn:aws:secretsmanager:us-east-1:111122223333:secret:appsync-
tutorial-rds-secret:*"
            ]
        }
    ]
}
```

Note that there are two statements in this policy to which you are granting role access. The first resource is your Aurora cluster and the second is your AWS Secrets Manager ARN.

Choose **Next**, review the configuration details, then choose **Create API**. You now have a fully operational API. You can review the full details of your API on the **Schema** page.

Resolvers for RDS

The API creation flow automatically created the resolvers to interact with our types. If you look at **Schema** page, you will find resolvers some of the follwoing resolvers.

- Create a `todo` via the `Mutation.createTodo` field.
- Update a `todo` via the `Mutation.updateTodo` field.
- Delete a `todo` via the `Mutation.deleteTodo` field.
- Get a single `todo` via the `Query.getTodo` field.
- List all `todos` via the `Query.listTodos` field.

You will find similar fields and resolvers attached for the `Task` type. Let's take a closer look at some of the resolvers.

Mutation.createTodo

From the schema editor in the AWS AppSync console, on the right side, choose `testdb` next to `createTodo(...): Todo`. The resolver code uses the `insert` function from the `rds` module to

dynamically create an insert statement that adds data to the `todos` table. Because we are working with Postgres, we can leverage the `returning` statement to get the inserted data back.

Update the following resolver to properly specify the `DATE` type of the `due` field.

```
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]
}
```

Save the resolver. The type hint marks the `due` properly in our input object as a `DATE` type. This allows the Postgres engine to properly interpret the value. Next, update your schema to remove the `id` from the `CreateTodo` input. Because our Postgres database can return the generated ID, you can rely on it for creation and returning the result as a single request as follows.

```
input CreateTodoInput {
    due: AWSDate!
```

```
    createdAt: String
    description: String!
}
```

Make the change and update your schema. Head to the **Queries** editor to add an item to the database as follows.

```
mutation CreateTodo {
  createTodo(input: {description: "Hello World!", due: "2023-12-31"}) {
    id
    due
    description
    createdAt
  }
}
```

You get the following result.

```
{
  "data": {
    "createTodo": {
      "id": 1,
      "due": "2023-12-31",
      "description": "Hello World!",
      "createdAt": "2023-11-14 20:47:11.875428"
    }
  }
}
```

Query.listTodos

From the schema editor in the console, on the right side, choose `testdb` next to `listTodos(id: ID!): Todo`. The request handler uses the select utility function to build a request dynamically at run time.

```
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)
}
```

We want to filter `todos` based on the due date. Let's update the resolver to cast due values to `DATE`. Update the list of imports and the request handler as follows.

```
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 };
}
```

In the **Queries** editor do the following.

```
query LIST {
  listTodos(limit: 10, filter: {due: {between: ["2021-01-01", "2025-01-02"]}}) {
    items {
      id
      due
      description
    }
  }
}
```

Mutation.updateTodo

You can also `update` a `Todo`. From the **Queries** editor, let's update our first `Todo` item of `id` `1`.

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits"}) {
    description
    due
    id
  }
}
```

Note that you must specify the `id` of the item you are updating. You can also specify a condition to only update an item that meets specific conditions. For example, we may only want to edit the item if the description starts with `edits` as follows.

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits: make a change"}, condition:
 {description: {beginsWith: "edits"}}) {
```

```
    description
    due
    id
  }
}
```

Just like how we handled our `create` and `list` operations, we can update our resolver to cast the due field to a `DATE`. Save these changes to `updateTodo` as follows.

```
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];
}
```

Now try an update with a condition:

```
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

You can `delete` a `Todo` with the `deleteTodo` mutation. This works like the `updateTodo` mutation, and you must specify the `id` of the item you want to delete as follows.

```
mutation DELETE {
  deleteTodo(input: {id: 1}) {
    description
    due
    id
  }
}
```

Writing custom queries

We've used the `rds` module utilities to create our SQL statements. We can also write our own custom static statement to interact with our database. First, update the schema to remove the `id` field from the `CreateTask` input.

```
input CreateTaskInput {
```

```
    todoId: Int!
    description: String
}
```

Next, create a couple of tasks. A task has a foreign key relationship with Todoas follows.

```
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 }
}
```

Create a new field in your `Query` type called `getTodoAndTasks`as follows.

```
getTodoAndTasks(id: Int!): Todo
```

Add a `tasks` field to the Todo type as follows.

```
type Todo {
    due: AWSDate!
    id: Int!
    createdAt: String
    description: String!
    tasks:TaskConnection
}
```

Save the schema. From the schema editor in the console, on the right side, choose **Attach Resolver** for `getTodosAndTasks(id: Int!): Todo`. Choose your Amazon RDS data source. Update your resolver with the following code.

```
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;
}
```

In this code, we use the `sql` tag template to write a SQL statement that we can safely pass a dynamic value to at run time. `createPgStatement` can take up to two SQL requests at a time. We use that to send one query for our `todo` and another for our `tasks`. You could have done this with a `JOIN` statement or any other method for that matter. The idea is being able to write your own SQL statement to implement your business logic. To use the query in the **Queries** editor, do the following.

```
query TodoAndTasks {
  getTodosAndTasks(id: 2) {
    id
    due
    description
    tasks {
      items {
        id
        description
      }
    }
  }
}
```

Deleting your cluster

> **Important**
>
> Deleting a cluster is permanent. Review your project thoroughly before carrying out this action.

To delete your cluster:

```
$ aws rds delete-db-cluster \
    --db-cluster-identifier appsync-tutorial \
```

```
    --skip-final-snapshot
```

VTL resolver tutorials for AWS AppSync

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

Data sources and resolvers are used by AWS AppSync to translate GraphQL requests and fetch information from your AWS resources. AWS AppSync supports automatic provisioning and connections with certain data source types. AWS AppSync also supports AWS Lambda, Amazon DynamoDB, relational databases (Amazon Aurora Serverless), Amazon OpenSearch Service, and HTTP endpoints as data sources. You can use a GraphQL API with your existing AWS resources or build data sources and resolvers from scratch. The following sections are meant to elucidate some of the more common GraphQL use cases in the form of tutorials.

AWS AppSync uses *mapping templates* written in Apache Velocity Template Language (VTL) for resolvers. For more information about using mapping templates, see the Resolver mapping template reference. More information about working with VTL is available in the Resolver mapping template programming guide.

AWS AppSync supports the automatic provisioning of DynamoDB tables from a GraphQL schema as described in Provision from schema (optional) and Launch a sample schema. You can also import from an existing DynamoDB table which will create schema and connect resolvers. This is outlined in Import from Amazon DynamoDB (optional).

Topics

- Using Aurora Serverless v2 with AWS AppSync
- Using pipeline resolvers in AWS AppSync
- Using Delta Sync operations on versioned data sources in AWS AppSync

Creating a simple post application using DynamoDB resolvers

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

This tutorial shows how you can bring your own Amazon DynamoDB tables to AWS AppSync and connect them to a GraphQL API.

You can let AWS AppSync provision DynamoDB resources on your behalf. Or, if you prefer, you can connect your existing tables to a GraphQL schema by creating a data source and a resolver. In either case, you'll be able to read and write to your DynamoDB database through GraphQL statements and subscribe to real-time data.

There are specific configuration steps that need to be completed in order for GraphQL statements to be translated to DynamoDB operations, and for responses to be translated back into GraphQL. This tutorial outlines the configuration process through several real-world scenarios and data access patterns.

Setting up your DynamoDB tables

To begin this tutorial, first you need to follow the steps below to provision AWS resources.

1. Provision AWS resources using the following AWS CloudFormation template in the CLI:

```
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
```

Alternatively, you can launch the following AWS CloudFormation stack in the US-West 2 (Oregon) region in your AWS account.

This creates the following:

- A DynamoDB table called `AppSyncTutorial-Post` that will hold `Post` data.
- An IAM role and associated IAM managed policy to allow AWS AppSync to interact with the `Post` table.

2. To see more details about the stack and the created resources, run the following CLI command:

```
aws cloudformation describe-stacks --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

3. To delete the resources later, you can run the following:

```
aws cloudformation delete-stack --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

Creating your GraphQL API

To create the GraphQL API in AWS AppSync:

1. Sign in to the AWS Management Console and open the AppSync console.
   - In the **APIs dashboard**, choose **Create API**.
2. Under the **Customize your API or import from Amazon DynamoDB** window, choose **Build from scratch**.
   - Choose **Start** to the right of the same window.
3. In the **API name** field, set the name of the API to `AWSAppSyncTutorial`.
4. Choose **Create**.

The AWS AppSync console creates a new GraphQL API for you using the API key authentication mode. You can use the console to set up the rest of the GraphQL API and run queries against it for the rest of this tutorial.

Defining a basic post API

Now that you have created an AWS AppSync GraphQL API, you can set up a basic schema that allows the basic creation, retrieval, and deletion of post data.

1. Sign in to the AWS Management Console and open the AppSync console.
   - In the **APIs dashboard**, choose the API you just created.
2. In the **Sidebar**, choose **Schema**.
   - In the **Schema** pane, replace the contents with the following code:

```
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. Choose **Save**.

This schema defines a `Post` type and operations to add and get `Post` objects.

Configuring the Data Source for the DynamoDB Tables

Next, link the queries and mutations defined in the schema to the `AppSyncTutorial-PostDynamoDB` table.

First, AWS AppSync needs to be aware of your tables. You do this by setting up a data source in AWS AppSync:

1. Sign in to the AWS Management Console and open the AppSync console.
   a. In the **APIs dashboard**, choose your GraphQL API.
   b. In the **Sidebar**, choose **Data Sources**.
2. Choose **Create data source**.
   a. For **Data source name**, enter in `PostDynamoDBTable`.
   b. For **Data source type**, choose **Amazon DynamoDB table**.
   c. For **Region**, choose **US-WEST-2**.
   d. For **Table name**, choose the **AppSyncTutorial-Post** DynamoDB table.
   e. Create a new IAM role (recommended) or choose an existing role that has the `lambda:invokeFunction` IAM permission. Existing roles need a trust policy, as explained in the Attaching a data source section.

      The following is an example IAM policy that has the required permissions to perform operations on the resource:

      JSON

```
{
     "Version": "2012-10-17",
     "Statement": [
         {
             "Effect": "Allow",
             "Action": [ "lambda:invokeFunction" ],
             "Resource": [
                 "arn:aws:lambda:us-
east-1:111122223333:function:myFunction",
```

```
                "arn:aws:lambda:us-
east-1:111122223333:function:myFunction:*"
            ]
        }
    ]
}
```

3. Choose **Create**.

Setting up the addPost resolver (DynamoDB PutItem)

After AWS AppSync is aware of the DynamoDB table, you can link it to individual queries and mutations by defining **Resolvers**. The first resolver you create is the `addPost` resolver, which enables you to create a post in the `AppSyncTutorial-Post` DynamoDB table.

A resolver has the following components:

- The location in the GraphQL schema to attach the resolver. In this case, you are setting up a resolver on the `addPost` field on the `Mutation` type. This resolver will be invoked when the caller calls `mutation { addPost(...){...} }`.
- The data source to use for this resolver. In this case, you want to use the `PostDynamoDBTable` data source you defined earlier, so you can add entries into the `AppSyncTutorial-Post` DynamoDB table.
- The request mapping template. The purpose of the request mapping template is to take the incoming request from the caller and translate it into instructions for AWS AppSync to perform against DynamoDB.
- The response mapping template. The job of the response mapping template is to take the response from DynamoDB and translate it back into something that GraphQL expects. This is useful if the shape of the data in DynamoDB is different to the `Post` type in GraphQL, but in this case they have the same shape, so you just pass the data through.

To set up the resolver:

1. Sign in to the AWS Management Console and open the AppSync console.
   a. In the **APIs dashboard**, choose your GraphQL API.
   b. In the **Sidebar**, choose **Data Sources**.
2. Choose **Create data source**.

a. For **Data source name**, enter in `PostDynamoDBTable`.

b. For **Data source type**, choose **Amazon DynamoDB table**.

c. For **Region**, choose **US-WEST-2**.

d. For **Table name**, choose the **AppSyncTutorial-Post** DynamoDB table.

e. Create a new IAM role (recommended) or choose an existing role that has the `lambda:invokeFunction` IAM permission. Existing roles need a trust policy, as explained in the Attaching a data source section.

The following is an example IAM policy that has the required permissions to perform operations on the resource:

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. Choose **Create**.
4. Choose the **Schema** tab.
5. In the **Data types** pane on the right, find the **addPost** field on the **Mutation** type, and then choose **Attach**.
6. In the **Action menu**, choose **Update runtime**, then choose **Unit Resolver (VTL only)**.
7. In **Data source name**, choose **PostDynamoDBTable**.
8. In **Configure the request mapping template**, paste the following:

```
{
    "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 }
    }
}
```

Note: A *type* is specified on all the keys and attribute values. For example, you set the `author` field to `{ "S" : "${context.arguments.author}" }`. The `S` part indicates to AWS AppSync and DynamoDB that the value will be a string value. The actual value gets populated from the `author` argument. Similarly, the `version` field is a number field because it uses `N` for the type. Finally, you're also initializing the `ups`, `downs` and `version` field.

For this tutorial you've specified that the GraphQL `ID!` type, which indexes the new item that is inserted to DynamoDB, comes as part of the client arguments. AWS AppSync comes with a utility for automatic ID generation called `$utils.autoId()` which you could have also used in the form of `"id" : { "S" : "${$utils.autoId()}" }`. Then you could simply leave the `id: ID!` out of the schema definition of `addPost()` and it would be inserted automatically. You won't use this technique for this tutorial, but you should consider it as a good practice when writing to DynamoDB tables.

For more information about mapping templates, see the Resolver Mapping Template Overview reference documentation. For more information about GetItem request mapping, see the GetItem reference documentation. For more information about types, see the Type System (Request Mapping) reference documentation.

9. In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

Note: Because the shape of the data in the `AppSyncTutorial-Post` table exactly matches the shape of the `Post` type in GraphQL, the response mapping template just passes the results straight through. Also note that all of the examples in this tutorial use the same response mapping template, so you only create one file.

10. Choose **Save**.

Call the API to Add a Post

Now that the resolver is set up, AWS AppSync can translate an incoming `addPost` mutation to a DynamoDB PutItem operation. You can now run a mutation to put something in the table.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following mutation:

```
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
  }
}
```

- Choose **Execute query** (the orange play button).
- The results of the newly created post should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

Here's what happened:

- AWS AppSync received an `addPost` mutation request.
- AWS AppSync took the request, and the request mapping template, and generated a request mapping document. This would have looked like:

```
{
    "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 used the request mapping document to generate and execute a DynamoDB`PutItem` request.
- AWS AppSync took the results of the `PutItem` request and converted them back to GraphQL types.

```
{
    "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
}
```

- Passed it through the response mapping document, which just passed it through unchanged.
- Returned the newly created object in the GraphQL response.

Setting Up the getPost Resolver (DynamoDB GetItem)

Now that you're able to add data to the `AppSyncTutorial-Post`DynamoDB table, you need to set up the `getPost` query so it can retrieve that data from the `AppSyncTutorial-Post` table. To do this, you set up another resolver.

- Choose the **Schema** tab.
- In the **Data types** pane on the right, find the **getPost** field on the **Query** type, and then choose **Attach**.
- In the **Action menu**, choose **Update runtime**, then choose **Unit Resolver (VTL only)**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

- Choose **Save**.

Call the API to Get a Post

Now the resolver has been set up, AWS AppSync knows how to translate an incoming `getPost` query to a DynamoDB`GetItem` operation. You can now run a query to retrieve the post you created earlier.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following:

```
query getPost {
  getPost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choose **Execute query** (the orange play button).
- The post retrieved from DynamoDB should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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
    }
  }
```

```
}
```

Here's what happened:

- AWS AppSync received a `getPost` query request.
- AWS AppSync took the request, and the request mapping template, and generated a request mapping document. This would have looked like:

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "123" }
    }
}
```

- AWS AppSync used the request mapping document to generate and execute a DynamoDB GetItem request.
- AWS AppSync took the results of the `GetItem` request and converted it back to GraphQL types.

```
{
    "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
}
```

- Passed it through the response mapping document, which just passed it through unchanged.
- Returned the retrieved object in the response.

Alternatively, take the following example:

```
query getPost {
  getPost(id:123) {
    id
```

```
    author
    title
  }
}
```

If your `getPost` query only needs the `id`, `author`, and `title`, you can change your request mapping template to use projection expressions to specify only the attributes that you want from your DynamoDB table to avoid unnecessary data transfer from DynamoDB to AWS AppSync. For example, the request mapping template may look like the snippet below:

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "projection" : {
     "expression" : "#author, id, title",
     "expressionNames" : { "#author" : "author"}
    }
}
```

Create an updatePost Mutation (DynamoDB UpdateItem)

So far you can create and retrieve `Post` objects in DynamoDB. Next, you'll set up a new mutation to allow us to update object. You'll do this using the UpdateItem DynamoDB operation.

- Choose the **Schema** tab.
- In the **Schema** pane, modify the `Mutation` type to add a new `updatePost` mutation as follows:

```
type Mutation {
    updatePost(
        id: ID!,
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post
    addPost(
        author: String!
        title: String!
```

```
        content: String!
        url: String!
    ): Post!
}
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **updatePost** field on the **Mutation** type and then choose **Attach**.
- In the **Action menu**, choose **Update runtime**, then choose **Unit Resolver (VTL only)**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
    "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 }
        }
    }
}
```

Note: This resolver is using the DynamoDB UpdateItem, which is significantly different from the PutItem operation. Instead of writing the entire item, you're just asking DynamoDB to update certain attributes. This is done using DynamoDB Update Expressions. The expression itself is specified in the `expression` field in the `update` section. It says to set the `author`, `title`, `content` and url attributes, and then increment the `version` field. The values to use do not appear in the expression itself; the expression has placeholders that have names starting with a

colon, which are then defined in the `expressionValues` field. Finally, DynamoDB has reserved words that cannot appear in the `expression`. For example, `url` is a reserved word, so to update the `url` field you can use name placeholders and define them in the `expressionNames` field.

For more info about `UpdateItem` request mapping, see the UpdateItem reference documentation. For more information about how to write update expressions, see the DynamoDB UpdateExpressions documentation.

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

Call the API to Update a Post

Now the resolver has been set up, AWS AppSync knows how to translate an incoming `update` mutation to a DynamoDB`Update` operation. You can now run a mutation to update the item you wrote earlier.

- Choose the **Queries** tab.
- In **Queries** pane, paste the following mutation. You'll also need to update the `id` argument to the value you noted down earlier.

```
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
  }
}
```

- Choose **Execute query** (the orange play button).
- The updated post in DynamoDB should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

In this example, the `ups` and `downs` fields were not modified because the request mapping template did not ask AWS AppSync and DynamoDB to do anything with those fields. Also, the `version` field was incremented by 1 because you asked AWS AppSync and DynamoDB to add 1 to the `version` field.

Modifying the updatePost Resolver (DynamoDB UpdateItem)

This is a good start to the `updatePost` mutation, but it has two main problems:

- If you want to update just a single field, you have to update all of the fields.
- If two people are modifying the object, you could potentially lose information.

To address these issues, you're going to modify the `updatePost` mutation to only modify arguments that were specified in the request, and then add a condition to the `UpdateItem` operation.

1. Choose the **Schema** tab.
2. In the **Schema** pane, modify the `updatePost` field in the `Mutation` type to remove the exclamation marks from the `author`, `title`, `content`, and `url` arguments, making sure

to leave the `id` field as is. This will make them optional argument. Also, add a new, required `expectedVersion` argument.

```
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. Choose **Save**.
4. In the **Data types** pane on the right, find the **updatePost** field on the **Mutation** type.
5. Choose **PostDynamoDBTable** to open the existing resolver.
6. In **Configure the request mapping template**, modify the request mapping template as follows:

```
{
    "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. Choose **Save**.

This template is one of the more complex examples. It demonstrates the power and flexibility of mapping templates. It loops through all of the arguments, skipping over `id` and `expectedVersion`. If the argument is set to something, it asks AWS AppSync and DynamoDB to update that attribute on the object in DynamoDB. If the attribute is set to null, it asks AWS

AppSync and DynamoDB to remove that attribute from the post object. If an argument wasn't specified, it leaves the attribute alone. It also increments the `version` field.

Also, there is a new `condition` section. A condition expression enables you tell AWS AppSync and DynamoDB whether or not the request should succeed based on the state of the object already in DynamoDB before the operation is performed. In this case, you only want the `UpdateItem` request to succeed if the `version` field of the item currently in DynamoDB exactly matches the `expectedVersion` argument.

For more information about condition expressions, see the Condition Expressions reference documentation.

Call the API to Update a Post

Let's try updating the `Post` object with the new resolver:

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following mutation. You'll also need to update the `id` argument to the value you noted down earlier.

```
mutation updatePost {
  updatePost(
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 2
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choose **Execute query** (the orange play button).
- The updated post in DynamoDB should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

In this request, you asked AWS AppSync and DynamoDB to update the `title` and `content` field only. It left all the other fields alone (other than incrementing the `version` field). You set the `title` attribute to a new value, and removed the `content` attribute from the post. The `author`, `url`, ups, and downs fields were left untouched.

Try executing the mutation request again, leaving the request exactly as is. You should see a response similar to the following:

```
{
  "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)"
    }
  ]
}
```

The request fails because the condition expression evaluates to false:

- The first time you ran the request, the value of the `version` field of the post in DynamoDB was 2, which matched the `expectedVersion` argument. The request succeeded, which meant the `version` field was incremented in DynamoDB to 3.
- The second time you ran the request, the value of the `version` field of the post in DynamoDB was 3, which did not match the `expectedVersion` argument.

This pattern is typically called *optimistic locking*.

A feature of an AWS AppSync DynamoDB resolver is that it returns the current value of the post object in DynamoDB. You can find this in the `data` field in the `errors` section of the GraphQL response. Your application can use this information to decide how it should proceed. In this case, you can see the `version` field of the object in DynamoDB is set to 3, so you could just update the `expectedVersion` argument to 3 and the request would succeed again.

For more information about handling condition check failures, see the Condition Expressions mapping template reference documentation.

Create upvotePost and downvotePost Mutations (DynamoDB UpdateItem)

The `Post` type has `ups` and `downs` fields to enable record upvotes and downvotes, but so far the API doesn't let us do anything with them. Let's add some mutations to let us upvote and downvote the posts.

- Choose the **Schema** tab.
- In the **Schema** pane, modify the `Mutation` type to add new `upvotePost` and `downvotePost` mutations as follows:

```
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!
}
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **upvotePost** field on the **Mutation** type, and then choose **Attach**.
- In the **Action menu**, choose **Update runtime**, then choose **Unit Resolver (VTL only)**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
    "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 }
        }
```

```
    }
}
```

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created `downvotePost` field on the **Mutation** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
    "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 }
        }
    }
}
```

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

- Choose **Save**.

Call the API to upvote and downvote a Post

Now the new resolvers have been set up, AWS AppSync knows how to translate an incoming `upvotePost` or `downvote` mutation to DynamoDB UpdateItem operation. You can now run mutations to upvote or downvote the post you created earlier.

- Choose the **Queries** tab.

- In the **Queries** pane, paste the following mutation. You'll also need to update the `id` argument to the value you noted down earlier.

```
mutation votePost {
  upvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choose **Execute query** (the orange play button).
- The post is updated in DynamoDB and should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

- Choose **Execute query** a few more times. You should see the `ups` and `version` field incrementing by 1 each time you execute the query.
- Change the query to call the `downvotePost` mutation as follows:

```
mutation votePost {
  downvotePost(id:123) {
    id
```

```
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choose **Execute query** (the orange play button). This time, you should see the `downs` and `version` field incrementing by 1 each time you execute the query.

```
{
  "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
    }
  }
}
```

Setting Up the deletePost Resolver (DynamoDB DeleteItem)

The next mutation you want to set up is to delete a post. You'll do this using the `DeleteItem` DynamoDB operation.

- Choose the **Schema** tab.
- In the **Schema** pane, modify the `Mutation` type to add a new `deletePost` mutation as follows:

```
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!
}
```

This time you made the `expectedVersion` field optional, which is explained later when you add the request mapping template.

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **delete** field on the **Mutation** type, and then choose **Attach**.
- In the **Action menu**, choose **Update runtime**, then choose **Unit Resolver (VTL only)**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
    "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
```

```
}
```

Note: The `expectedVersion` argument is an optional argument. If the caller set an `expectedVersion` argument in the request, the template adds a condition that only allows the `DeleteItem` request to succeed if the item is already deleted or if the `version` attribute of the post in DynamoDB exactly matches the `expectedVersion`. If left out, no condition expression is specified on the `DeleteItem` request. It succeeds regardless of the value of `version`, or whether or not the item exists in DynamoDB.

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

Note: Even though you're deleting an item, you can return the item that was deleted, if it was not already deleted.

- Choose **Save**.

For more info about `DeleteItem` request mapping, see the DeleteItem reference documentation.

Call the API to Delete a Post

Now the resolver has been set up, AWS AppSync knows how to translate an incoming `delete` mutation to a DynamoDB`DeleteItem` operation. You can now run a mutation to delete something in the table.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following mutation. You'll also need to update the `id` argument to the value you noted down earlier.

```
mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
```

```
  }
}
```

- Choose **Execute query** (the orange play button).
- The post is deleted from DynamoDB. Note that AWS AppSync returns the value of the item that was deleted from DynamoDB, which should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

The value is only returned if this call to `deletePost` was the one that actually deleted it from DynamoDB.

- Choose **Execute query** again.
- The call still succeeds, but no value is returned.

```
{
  "data": {
    "deletePost": null
  }
}
```

Now let's try deleting a post, but this time specifying an `expectedValue`. First though, you'll need to create a new post because you've just deleted the one you've been working with so far.

- In the **Queries** pane, paste the following mutation:

```
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
  }
}
```

- Choose **Execute query** (the orange play button).
- The results of the newly created post should appear in the results pane to the right of the query pane. Note down the `id` of the newly created object because you need it in just a moment. It should look similar to the following:

```
{
  "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
    }
  }
}
```

Now let's try to delete that post, but put in the wrong value for `expectedVersion`:

- In the **Queries** pane, paste the following mutation. You'll also need to update the `id` argument to the value you noted down earlier.

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choose **Execute query** (the orange play button).

```
{
  "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)"
    }
  ]
}
```

The request failed because the condition expression evaluates to false: the value for `version` of the post in DynamoDB does not match the `expectedValue` specified in the arguments. The current value of the object is returned in the `data` field in the `errors` section of the GraphQL response.

- Retry the request, but correct the `expectedVersion`:

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choose **Execute query** (the orange play button).
- This time the request succeeds, and the value that was deleted from DynamoDB is returned:

```
{
  "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
    }
  }
}
```

- Choose **Execute query** again.
- The call still succeeds, but this time no value is returned because the post was already deleted in DynamoDB.

```
{
  "data": {
    "deletePost": null
  }
}
```

Setting Up the allPost Resolver (DynamoDB Scan)

So far the API is only useful if you know the `id` of each post you want to look at. Let's add a new resolver that returns all the posts in the table.

- Choose the **Schema** tab.
- In the **Schema** pane, modify the `Query` type to add a new `allPost` query as follows:

```
type Query {
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

- Add a new `PaginationPosts` type:

```
type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **allPost** field on the **Query** type, and then choose **Attach**.
- In the **Action menu**, choose **Update runtime**, then choose **Unit Resolver (VTL only)**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
    "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
}
```

  This resolver has two optional arguments: `count`, which specifies the maximum number of items to return in a single call, and `nextToken`, which can be used to retrieve the next set of results (you'll show where the value for `nextToken` comes from later).

- In **Configure the response mapping template**, paste the following:

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

  Note: This response mapping template is different from all the others so far. The result of the `allPost` query is a `PaginatedPosts`, which contains a list of posts and a pagination token. The shape of this object is different to what is returned from the AWS AppSync DynamoDB Resolver: the list of posts is called `items` in the AWS AppSync DynamoDB Resolver results, but is called `posts` in `PaginatedPosts`.

- Choose **Save**.

For more information about `Scan` request mapping, see the Scan reference documentation.

Call the API to Scan All Posts

Now the resolver has been set up, AWS AppSync knows how to translate an incoming `allPost` query to a DynamoDB`Scan` operation. You can now scan the table to retrieve all the posts.

Before you can try it out though, you need to populate the table with some data because you've deleted everything you've worked with so far.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following mutation:

```
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 }
}
```

- Choose **Execute query** (the orange play button).

Now, let's scan the table, returning five results at a time.

- In the **Queries** pane, paste the following query:

```
query allPost {
  allPost(count: 5) {
```

```
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).
- The first five posts should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

You got five results and a `nextToken` that you can use to get the next set of results.

- Update the `allPost` query to include the `nextToken` from the previous set of results:

```
query allPost {
  allPost(
    count: 5
    nextToken:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
  ) {
    posts {
      id
      author
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).
- The remaining four posts should appear in the results pane to the right of the query pane. There is no `nextToken` in this set of results because you've paged through all nine posts, with none remaining. It should look similar to the following:

```
{
  "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
    }
  }
}
```

Setting Up the allPostsByAuthor Resolver (DynamoDB Query)

In addition to scanning DynamoDB for all posts, you can also query DynamoDB to retrieve posts created by a specific author. The DynamoDB table you created earlier already has a `GlobalSecondaryIndex` called `author-index` you can use with a DynamoDB`Query` operation to retrieve all posts created by a specific author.

- Choose the **Schema** tab.
- In the **Schema** pane, modify the `Query` type to add a new `allPostsByAuthor` query as follows:

```
type Query {
    allPostsByAuthor(author: String!, count: Int, nextToken: String): PaginatedPosts!
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

  Note: This uses the same `PaginatedPosts` type that you used with the `allPost` query.
- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **allPostsByAuthor** field on the **Query** type, and then choose **Attach**.
- In the **Action menu**, choose **Update runtime**, then choose **Unit Resolver (VTL only)**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
    "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
}
```

Like the `allPost` resolver, this resolver has two optional arguments: `count`, which specifies the maximum number of items to return in a single call, and `nextToken`, which can be used to retrieve the next set of results (the value for `nextToken` can be obtained from a previous call).

- In **Configure the response mapping template**, paste the following:

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

Note: This is the same response mapping template that you used in the `allPost` resolver.

- Choose **Save**.

For more information about `Query` request mapping, see the Query reference documentation.

Call the API to Query All Posts by an Author

Now the resolver has been set up, AWS AppSync knows how to translate an incoming `allPostsByAuthor` mutation to a DynamoDB`Query` operation against the `author-index` index. You can now query the table to retrieve all the posts by a specific author.

Before you do that, however, let's populate the table with some more posts, because every post so far has the same author.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following mutation:

```
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 }
}
```

- Choose **Execute query** (the orange play button).

Now, let's query the table, returning all posts authored by `Nadia`.

- In the **Queries** pane, paste the following query:

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).
- All the posts authored by `Nadia` should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
```

```
        }
      ],
      "nextToken": null
    }
  }
}
```

Pagination works for `Query` just the same as it does for `Scan`. For example, let's look for all posts by `AUTHORNAME`, getting five at a time.

- In the **Queries** pane, paste the following query:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).
- All the posts authored by `AUTHORNAME` should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

- Update the `nextToken` argument with the value returned from the previous query as follows:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
    nextToken:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).
- The remaining posts authored by `AUTHORNAME` should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

Using Sets

Up to this point the `Post` type has been a flat key/value object. You can also model complex objects with the AWS AppSyncDynamoDB resolver, such as sets, lists, and maps.

Let's update the `Post` type to include tags. A post can have 0 or more tags, which are stored in DynamoDB as a String Set. You'll also set up some mutations to add and remove tags, and a new query to scan for posts with a specific tag.

- Choose the **Schema** tab.
- In the **Schema** pane, modify the `Post` type to add a new `tags` field as follows:

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
```

```
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
}
```

- In the **Schema** pane, modify the `Query` type to add a new `allPostsByTag` query as follows:

```
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
}
```

- In the **Schema** pane, modify the `Mutation` type to add new `addTag` and `removeTag` mutations as follows:

```
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!
}
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **allPostsByTag** field on the **Query** type, and then choose **Attach**.

- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
    "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
}
```

- In **Configure the response mapping template**, paste the following:

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **addTag** field on the **Mutation** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
    "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 }
        }
    }
}
```

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **removeTag** field on the **Mutation** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
    "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 }
        }
    }
}
```

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

- Choose **Save**.

Call the API to Work with Tags

Now that you've set up the resolvers, AWS AppSync knows how to translate incoming `addTag`, `removeTag`, and `allPostsByTag` requests into DynamoDB `UpdateItem` and `Scan` operations.

To try it out, let's select one of the posts you created earlier. For example, let's use a post authored by `Nadia`.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following query:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).
- All of Nadia's posts should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
    }
```

```
  }
}
```

- Let's use the one with the title `"The cutest dog in the world"`. Note down its `id` because you'll use it later.

Now let's try adding a dog tag.

- In the **Queries** pane, paste the following mutation. You'll also need to update the `id` argument to the value you noted down earlier.

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

- Choose **Execute query** (the orange play button).
- The post is updated with the new tag.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

You can add more tags as follows:

- Update the mutation to change the `tag` argument to puppy.

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
```

```
    id
    title
    tags
  }
}
```

- Choose **Execute query** (the orange play button).
- The post is updated with the new tag.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

You can also delete tags:

- In the **Queries** pane, paste the following mutation. You'll also need to update the `id` argument to the value you noted down earlier.

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- Choose **Execute query** (the orange play button).
- The post is updated and the puppy tag is deleted.

```
{
  "data": {
    "addTag": {
```

```
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

You can also search for all posts that have a tag:

- In the **Queries** pane, paste the following query:

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).
- All posts that have the dog tag are returned as follows:

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
            "dog",
            "puppy"
          ]
        }
      ],
      "nextToken": null
    }
```

```
  }
}
```

Using Lists and Maps

In addition to using DynamoDB sets, you can also use DynamoDB lists and maps to model complex data in a single object.

Let's add the ability to add comments to posts. This will be modeled as a list of map objects on the `Post` object in DynamoDB.

Note: in a real application, you would model comments in their own table. For this tutorial, you'll just add them in the `Post` table.

- Choose the **Schema** tab.
- In the **Schema** pane, add a new `Comment` type as follows:

```
type Comment {
    author: String!
    comment: String!
}
```

- In the **Schema** pane, modify the `Post` type to add a new `comments` field as follows:

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
  comments: [Comment!]
}
```

- In the **Schema** pane, modify the `Mutation` type to add a new `addComment` mutation as follows:

```
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!
}
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **addComment** field on the **Mutation** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
  "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)
    }
  }
}
```

This update expression will append a list containing our new comment to the existing `comments` list. If the list doesn't already exist, it will be created.

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

- Choose **Save**.

Call the API to Add a Comment

Now that you've set up the resolvers, AWS AppSync knows how to translate incoming `addComment` requests into DynamoDB`UpdateItem` operations.

Let's try it out by adding a comment to the same post you added the tags to.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following query:

```
mutation addComment {
  addComment(
    id:10
    author: "Steve"
    comment: "Such a cute dog."
  ) {
    id
    comments {
      author
      comment
    }
  }
}
```

- Choose **Execute query** (the orange play button).

- All of Nadia's posts should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "data": {
    "addComment": {
      "id": "10",
      "comments": [
        {
          "author": "Steve",
          "comment": "Such a cute dog."
        }
      ]
    }
  }
}
```

If you execute the request multiple times, multiple comments will be appended to the list.

Conclusion

In this tutorial, you've built an API that lets us manipulate Post objects in DynamoDB using AWS AppSync and GraphQL. For more information, see the Resolver Mapping Template Reference.

To clean up, you can delete the AppSync GraphQL API from the console.

To delete the DynamoDB table and the IAM role you created for this tutorial, you can run the following to delete the `AWSAppSyncTutorialForAmazonDynamoDB` stack, or visit the AWS CloudFormation console and delete the stack:

```
aws cloudformation delete-stack \
    --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

Using AWS Lambda resolvers in AWS AppSync

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

You can use AWS Lambda with AWS AppSync to resolve any GraphQL field. For example, a GraphQL query might send a call to an Amazon Relational Database Service (Amazon RDS) instance, and a GraphQL mutation might write to an Amazon Kinesis stream. In this section, we'll show you how to write a Lambda function that performs business logic based on the invocation of a GraphQL field operation.

Create a Lambda function

The following example shows a Lambda function written in `Node.js` that performs different operations on blog posts as part of a blog post application.

```
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;
    }
};
```

This Lambda function retrieves a post by ID, adds a post, retrieves a list of posts, and fetches related posts for a given post.

Note: The Lambda function uses the `switch` statement on `event.field` to determine which field is currently being resolved.

Create this Lambda function using the AWS Management Console or an AWS CloudFormation stack. To create the function from a CloudFormation stack, you can use the following AWS Command Line Interface (AWS CLI) command:

```
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
```

You can also launch the AWS CloudFormation stack in the US West (Oregon) AWS Region in your AWS account from here:

Configure a data source for Lambda

After you create the Lambda function, navigate to your GraphQL API in the AWS AppSync console, and then choose the **Data Sources** tab.

Choose **Create data source**, enter a friendly **Data source name** (for example, **Lambda**), and then for **Data source type**, choose **AWS Lambda function**. For **Region**, choose the same Region as your function. (If you created the function from the provided CloudFormation stack, the function is probably in **US-WEST-2**.) For **Function ARN**, choose the Amazon Resource Name (ARN) of your Lambda function.

After choosing your Lambda function, you can either create a new AWS Identity and Access Management (IAM) role (for which AWS AppSync assigns the appropriate permissions) or choose an existing role that has the following inline policy:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:us-
east-1:111122223333:function:LAMBDA_FUNCTION"
        }
    ]
}
```

You must also set up a trust relationship with AWS AppSync for the IAM role as follows:

JSON

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

Create a GraphQL schema

Now that the data source is connected to your Lambda function, create a GraphQL schema.

From the schema editor in the AWS AppSync console, make sure that your schema matches the following schema:

```
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]
}
```

Configure resolvers

Now that you've registered a Lambda data source and a valid GraphQL schema, you can connect your GraphQL fields to your Lambda data source using resolvers.

To create a resolver, you'll need mapping templates. To learn more about mapping templates, see Resolver Mapping Template Overview.

For more information about Lambda mapping templates, see Resolver mapping template reference for Lambda.

In this step, you attach a resolver to the Lambda function for the following fields: `getPost(id:ID!): Post`, `allPosts: [Post]`, `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`, and `Post.relatedPosts: [Post]`.

From the schema editor in the AWS AppSync console, on the right side, choose **Attach Resolver** for `getPost(id:ID!): Post`.

Then, in the **Action menu**, choose **Update runtime**, then choose **Unit Resolver (VTL only)**.

Afterward, choose your Lambda data source. In the **request mapping template** section, choose **Invoke And Forward Arguments**.

Modify the `payload` object to add the field name. Your template should look like the following:

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments":  $utils.toJson($context.arguments)
```

```
    }
}
```

In the **response mapping template** section, choose **Return Lambda Result**.

In this case, use the base template as-is. It should look like the following:

```
$utils.toJson($context.result)
```

Choose **Save**. You have successfully attached your first resolver. Repeat this operation for the remaining fields as follows:

For `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` request mapping template:

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "addPost",
        "arguments":  $utils.toJson($context.arguments)
    }
}
```

For `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` response mapping template:

```
$utils.toJson($context.result)
```

For `allPosts: [Post]` request mapping template:

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "allPosts"
    }
}
```

For `allPosts: [Post]` response mapping template:

```
$utils.toJson($context.result)
```

For `Post.relatedPosts: [Post]` request mapping template:

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "relatedPosts",
        "source":  $utils.toJson($context.source)
    }
}
```

For `Post.relatedPosts: [Post]` response mapping template:

```
$utils.toJson($context.result)
```

Test your GraphQL API

Now that your Lambda function is connected to GraphQL resolvers, you can run some mutations and queries using the console or a client application.

On the left side of the AWS AppSync console, choose **Queries**, and then paste in the following code:

addPost Mutation

```
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 Query

```
query getPost {
    getPost(id: "2") {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

allPosts Query

```
query allPosts {
    allPosts {
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {
            id
            title
        }
    }
}
```

Returning errors

Any given field resolution can result in an error. With AWS AppSync, you can raise errors from the following sources:

- Request or response mapping template
- Lambda function

From the mapping template

To raise intentional errors, you can use the `$utils.error` helper method from the Velocity Template Language (VTL) template. It takes as argument an `errorMessage`, an `errorType`, and an optional `data` value. The `data` is useful for returning extra data back to the client when an error occurs. The `data` object is added to the `errors` in the GraphQL final response.

The following example shows how to use it in the `Post.relatedPosts: [Post]` response mapping template:

```
$utils.error("Failed to fetch relatedPosts", "LambdaFailure", $context.result)
```

This yields a GraphQL response similar to the following:

```
{
    "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"
                }
            ]
        }
    ]
}
```

Where `allPosts[0].relatedPosts` is *null* because of the error and the `errorMessage`, `errorType`, and `data` are present in the `data.errors[0]` object.

From the Lambda function

AWS AppSync also understands errors that the Lambda function throws. The Lambda programming model lets you raise *handled* errors. If the Lambda function throws an error, AWS AppSync fails to resolve the current field. Only the error message returned from Lambda is set in the response. Currently, you can't pass any extraneous data back to the client by raising an error from the Lambda function.

Note: If your Lambda function raises an *unhandled* error, AWS AppSync uses the error message that Lambda set.

The following Lambda function raises an error:

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    callback("I fail. Always.");
};
```

This returns a GraphQL response similar to the following:

```
{
    "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."
        }
    ]
}
```

Advanced use case: Batching

The Lambda function in this example has a `relatedPosts` field that returns a list of related posts for a given post. In the example queries, the `allPosts` field invocation from the Lambda function returns five posts. Because we specified that we also want to resolve `relatedPosts` for each returned post, the `relatedPosts` field operation is invoked five times.

```
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
        }
    }
}
```

While this might not sound substantial in this specific example, this compounded over-fetching can quickly undermine the application.

If you were to fetch `relatedPosts` again on the returned related `Posts` in the same query, the number of invocations would increase dramatically.

```
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
            }
        }
    }
}
```

In this relatively simple query, AWS AppSync would invoke the Lambda function 1 + 5 + 25 = 31 times.

This is a fairly common challenge and is often called the N+1 problem (in this case, N = 5), and it can incur increased latency and cost to the application.

One approach to solving this issue is to batch similar field resolver requests together. In this example, instead of having the Lambda function resolve a list of related posts for a single given post, it could instead resolve a list of related posts for a given batch of posts.

To demonstrate this, let's switch the `Post.relatedPosts: [Post]` resolver to a batch-enabled resolver.

On the right side of the AWS AppSync console, choose the existing `Post.relatedPosts: [Post]` resolver. Change the request mapping template to the following:

```
{
    "version": "2017-02-28",
    "operation": "BatchInvoke",
    "payload": {
        "field": "relatedPosts",
        "source":  $utils.toJson($context.source)
    }
}
```

Only the `operation` field has changed from `Invoke` to `BatchInvoke`. The payload field now becomes an array of whatever is specified in the template. In this example, the Lambda function receives the following as input:

```
[
    {
        "field": "relatedPosts",
        "source": {
            "id": 1
        }
    },
    {
        "field": "relatedPosts",
        "source": {
            "id": 2
        }
    },
    ...
]
```

When `BatchInvoke` is specified in the request mapping template, the Lambda function receives a list of requests and returns a list of results.

Specifically, the list of results must match the size and order of the request payload entries so that AWS AppSync can match the results accordingly.

In this batching example, the Lambda function returns a batch of results as follows:

```
[
    [{"id":"2","title":"Second book"}, {"id":"3","title":"Third book"}],   //
 relatedPosts for id=1
    [{"id":"3","title":"Third book"}]
           // relatedPosts for id=2
]
```

The following Lambda function in Node.js demonstrates this batching functionality for the `Post.relatedPosts` field as follows:

```
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;
    }
};
```

Returning individual errors

The previous examples show that it's possible to return a single error from the Lambda function or raise an error from the mapping templates. For batched invocations, raising an error from the Lambda function flags an entire batch as failed. This might be acceptable for specific scenarios where an irrecoverable error occurs, such as a failed connection to a data store. However, in cases where some items in the batch succeed and others fail, it's possible to return both errors and valid data. Because AWS AppSync requires the batch response to list elements matching the original size of the batch, you must define a data structure that can differentiate valid data from an error.

For example, if the Lambda function is expected to return a batch of related posts, you could choose to return a list of `Response` objects where each object has optional *data*, *errorMessage*, and *errorType* fields. If the *errorMessage* field is present, it means that an error occurred.

The following code shows how you could update the Lambda function:

```
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;
```

```
    }
};
```

For this example, the following response mapping template parses each item of the Lambda function and raises any errors that occur:

```
#if( $context.result && $context.result.errorMessage )
    $utils.error($context.result.errorMessage, $context.result.errorType,
 $context.result.data)
#else
    $utils.toJson($context.result.data)
#end
```

This example returns a GraphQL response similar to the following:

```
{
  "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"
    }
  ]
}
```

Configuring the maximum batching size

By default, when using `BatchInvoke`, AWS AppSync sends requests to your Lambda function in batches of up to five items. You can configure the maximum batch size of your Lambda resolvers.

To configure the maximum batching size on a resolver, use the following command in the AWS Command Line Interface (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**
>
> When providing a request mapping template, you must use the `BatchInvoke` operation to use batching.

You can also use the following command to enable and configure batching on Direct Lambda Resolvers:

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
 relatedPosts \
 --data-source-name "<lambda-datasource>" \
 --max-batch-size X
```

Maximum batching size configuration with VTL templates

For Lambda Resolvers that have VTL in-request templates, the maximum batch size will have no effect unless they have directly specified it as a `BatchInvoke` operation in VTL. Similarly, if you are performing a top-level mutation, batching is not conducted for mutations because the GraphQL specification requires parallel mutations to be executed sequentially.

For example, take the following mutations:

```
type Mutation {
    putItem(input: Item): Item
    putItems(inputs: [Item]): [Item]
}
```

Using the first mutation, we can create 10 `Items` as shown in the snippet below:

```
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
    }
}
```

In this example, the `Items` will not be batched in a group of 10 even if the maximum batch size is set to 10 in the Lambda Resolver. Instead, they will execute sequentially according to the GraphQL specification.

To perform an actual batch mutation, you may follow the example below using the second mutation:

```
mutation MyMutation {
    putItems([$someItem1, $someItem2, $someItem3,$someItem4, $someItem5, $someItem6,
    $someItem7, $someItem8, $someItem9, $someItem10]) {
    id,
    name
    }
}
```

For more information about using batching with Direct Lambda Resolvers, see Direct Lambda Resolvers.

Using Amazon OpenSearch Service resolvers in AWS AppSync

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

AWS AppSync supports using Amazon OpenSearch Service from domains that you have provisioned in your own AWS account, provided they don't exist inside a VPC. After your domains are provisioned, you can connect to them using a data source, at which point you can configure a resolver in the schema to perform GraphQL operations such as queries, mutations, and subscriptions. This tutorial will take you through some common examples.

For more information, see the Resolver Mapping Template Reference for OpenSearch.

One-Click Setup

To automatically set up a GraphQL endpoint in AWS AppSync with Amazon OpenSearch Service configured you can use this AWS CloudFormation template:

After the AWS CloudFormation deployment completes you can skip directly to running GraphQL queries and mutations.

Create a New OpenSearch Service Domain

To get started with this tutorial, you need an existing OpenSearch Service domain. If you don't have one, you can use the following sample. Note that it can take up to 15 minutes for an OpenSearch Service domain to be created before you can move on to integrating it with an AWS AppSync data source.

```
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
```

You can launch the following AWS CloudFormation stack in the US West 2 (Oregon) region in your AWS account:

Configure Data Source for OpenSearch Service

After the OpenSearch Service domain is created, navigate to your AWS AppSync GraphQL API and choose the **Data Sources** tab. Choose **New** and enter a friendly name for the data source, such as “oss”. Then choose **Amazon OpenSearch domain** for **Data source type**, choose the appropriate region, and you should see your OpenSearch Service domain listed. After selecting it you can either create a new role and AWS AppSync will assign the role-appropriate permissions, or you can choose an existing role, which has the following inline policy:

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-1:111122223333:domain/democluster/*"
            ]
        }
    ]
}
```

You’ll also need to set up a trust relationship with AWS AppSync for that role:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Principal": {
                "Service": "appsync.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

Additionally, the OpenSearch Service domain has it's own **Access Policy** which you can modify through the Amazon OpenSearch Service console. You will need to add a policy similar to the following, with the appropriate actions and resource for the OpenSearch Service domain. Note that the **Principal** will be the AppSync data source role, which if you let the console create this, can be found in the IAM console.

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-1:111122223333:domain/DOMAIN_NAME/*"
        }
    ]
}
```

Connecting a Resolver

Now that the data source is connected to your OpenSearch Service domain, you can connect it to your GraphQL schema with a resolver, as shown in the following example:

```
 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
}
...
```

Note that there is a user-defined `Post` type with a field of `id`. In the following examples, we assume there is a process (which can be automated) for putting this type into your OpenSearch Service domain, which would map to a path root of `/post/_doc`, where `post` is the index. From this root path, you can perform individual document searches, wildcard searches with `/id/post*`, or multi-document searches with a path of `/post/_search`. For example, if you have another type called `User`, you can index documents under a new index called `user`, then perform searches with a **path** of `/user/_search`.

From the schema editor in the AWS AppSync console, modify the preceding `Posts` schema to include a `searchPosts` query:

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

Save the schema. On the right side, for `searchPosts`, choose **Attach resolver**. In the **Action menu**, choose **Update runtime**, then choose **Unit Resolver (VTL only)**. Then, choose your OpenSearch Service data source. Under the **request mapping template** section, select the drop-down for **Query posts** to get a base template. Modify the `path` to be `/post/_search`. It should look like the following:

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/post/_search",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "from":0,
            "size":50
        }
    }
}
```

This assumes that the preceding schema has documents that have been indexed in OpenSearch Service under the `post` field. If you structure your data differently, then you'll need to update accordingly.

Under the **response mapping template** section, you need to specify the appropriate `_source` filter if you want to get back the data results from an OpenSearch Service query and translate to GraphQL. Use the following template:

```
[
    #foreach($entry in $context.result.hits.hits)
    #if( $velocityCount > 1 ) , #end
    $utils.toJson($entry.get("_source"))
    #end
]
```

Modifying Your Searches

The preceding request mapping template performs a simple query for all records. Suppose you want to search by a specific author. Further, suppose you want that author to be an argument defined in your GraphQL query. In the schema editor of the AWS AppSync console, add an `allPostsByAuthor` query:

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

Now choose **Attach resolver** and select the OpenSearch Service data source, but use the following example in the **response mapping template**:

```
{
    "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)
                }
            }
        }
    }
}
```

Note that the body is populated with a term query for the `author` field, which is passed through from the client as an argument. You could optionally have prepopulated information, such as standard text, or even use other utilities.

If you're using this resolver, fill in the **response mapping template** with the same information as the previous example.

Adding Data to OpenSearch Service

You may want to add data to your OpenSearch Service domain as the result of a GraphQL mutation. This is a powerful mechanism for searching and other purposes. Because you can use GraphQL subscriptions to make your data real-time, it serves as a mechanism for notifying clients of updates to data in your OpenSearch Service domain.

Return to the **Schema** page in the AWS AppSync console and select **Attach resolver** for the `addPost()` mutation. Select the OpenSearch Service data source again and use the following **response mapping template** for the `Posts` schema:

```
{
    "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)
        }
    }
}
```

As before, this is an example of how your data might be structured. If you have different field names or indexes, you need to update the `path` and `body` as appropriate. This example also shows how to use `$context.arguments` to populate the template from your GraphQL mutation arguments.

Before moving on, use the following response mapping template, which will return the result of the mutation operation or error information as output:

```
#if($context.error)
    $util.toJson($ctx.error)
#else
    $util.toJson($context.result)
#end
```

Retrieving a Single Document

Finally, if you want to use the `getPost(id:ID)` query in your schema to return an individual document, find this query in the schema editor of the AWS AppSync console and choose **Attach resolver**. Select the OpenSearch Service data source again and use the following mapping template:

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path": $util.toJson("post/_doc/$context.arguments.id"),
    "params":{
        "headers":{},
        "queryString":{},
        "body":{}
    }
}
```

Because the `path` above uses the `id` argument with an empty body, this returns the single document. However, you need to use the following response mapping template, because now you're returning a single item and not a list:

```
$utils.toJson($context.result.get("_source"))
```

Perform Queries and Mutations

You should now be able to perform GraphQL operations against your OpenSearch Service domain. Navigate to the **Queries** tab of the AWS AppSync console and add a new record:

```
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
       )
}
```

You'll see the result of the mutation on the right. Similarly, you can now run a `searchPosts` query against your OpenSearch Service domain:

```
query searchPosts {
    searchPosts {
        id
        title
        author
        content
    }
}
```

Best Practices

- OpenSearch Service should be for querying data, not as your primary database. You may want to use OpenSearch Service in conjunction with Amazon DynamoDB as outlined in Combining GraphQL Resolvers.
- Only give access to your domain by allowing the AWS AppSync service role to access the cluster.
- You can start small in development, with the lowest-cost cluster, and then move to a larger cluster with high availability (HA) as you move into production.

Using local resolvers in AWS AppSync

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

AWS AppSync allows you to use supported data sources (AWS Lambda, Amazon DynamoDB, or Amazon OpenSearch Service) to perform various operations. However, in certain scenarios, a call to a supported data source might not be necessary.

This is where the local resolver comes in handy. Instead of calling a remote data source, the local resolver will just **forward** the result of the request mapping template to the response mapping template. The field resolution will not leave AWS AppSync.

Local resolvers are useful for several use cases. The most popular use case is to publish notifications without triggering a data source call. To demonstrate this use case, let's build a paging application; where users can page each other. This example leverages *Subscriptions*, so if you aren't familiar with *Subscriptions*, you can follow the Real-Time Data tutorial.

Create the Paging Application

In our paging application, clients can subscribe to an inbox, and send pages to other clients. Each page includes a message. Here is the schema:

```
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
```

```
}
```

Let's attach a resolver on the `Mutation.page` field. In the **Schema** pane, click on *Attach Resolver* next to the field definition on the right panel. Create a new data source of type *None* and name it *PageDataSource*.

For the request mapping template, enter:

```
{
  "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()"
  }
}
```

And for the response mapping template, select the default *Forward the result*. Save your resolver. You application is now ready, let's page!

Send and subscribe to pages

For clients to receive pages, they must first be subscribed to an inbox.

In the **Queries** pane let's execute the `inbox` subscription:

```
subscription Inbox {
    inbox(to: "Nadia") {
        body
        to
        from
        sentAt
    }
}
```

Nadia will receive pages whenever the `Mutation.page` mutation is invoked. Let's invoke the mutation by executing the mutation:

```
mutation Page {
    page(to: "Nadia", body: "Hello, World!") {
```

```
        body
        to
        from
        sentAt
    }
}
```

We just demonstrated the use of local resolvers, by sending a Page and receiving it without leaving AWS AppSync.

Combining GraphQL resolvers in AWS AppSync

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

Resolvers and fields in a GraphQL schema have 1:1 relationships with a large degree of flexibility. Because a data source is configured on a resolver independently of a schema, you have the ability for GraphQL types to be resolved or manipulated through different data sources, mixing and matching on a schema to best meet your needs.

The following example scenarios demonstrate how to mix and match data sources in your schema. Before you begin, we recommend that you are familiar with setting up data sources and resolvers for AWS Lambda, Amazon DynamoDB, and Amazon OpenSearch Service as described in the previous tutorials.

Example schema

The following schema has a type of `Post` with 3 `Query` operations and 3 `Mutation` operations defined:

```
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
}
```

In this example you would have a total of 6 resolvers to attach. One possible way would to have all of these come from an Amazon DynamoDB table, called `Posts`, where `AllPosts` runs a scan and `searchPosts` runs a query, as outlined in the DynamoDB Resolver Mapping Template Reference. However, there are alternatives to meet your business needs, such as having these GraphQL queries resolve from Lambda or OpenSearch Service.

Alter data through resolvers

You might have the need to return results from a database such as DynamoDB (or Amazon Aurora) to clients with some of the attributes changed. This might be due to formatting of the data types, such as timestamp differences on clients, or to handle backwards compatibility issues. For

illustrative purposes, in the following example, an AWS Lambda function manipulates the up-votes and down-votes for blog posts by assigning them random numbers each time the GraphQL resolver is invoked:

```
'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);
    });
};
```

This is a perfectly valid Lambda function and could be attached to the `AllPosts` field in the GraphQL schema so that any query returning all the results gets random numbers for the ups/downs.

DynamoDB and OpenSearch Service

For some applications, you might perform mutations or simple lookup queries against DynamoDB, and have a background process transfer documents to OpenSearch Service. You can then simply attach the `searchPosts` Resolver to the OpenSearch Service data source and return search results (from data that originated in DynamoDB) using a GraphQL query. This can be extremely powerful when adding advanced search operations to your applications such keyword, fuzzy word matches or even geospatial lookups. Transferring data from DynamoDB could be done through an ETL process or alternatively you can stream from DynamoDB using Lambda. You can launch a complete example of this using the following AWS CloudFormation stack in the US West 2 (Oregon) Region in your AWS account:

The schema in this example lets you add posts using a DynamoDB resolver as follows:

```
mutation add {
    putPost(author:"Nadia"
        title:"My first post"
        content:"This is some test content"
        url:"https://aws.amazon.com/appsync/"
    ){
        id
        title
    }
}
```

This writes data to DynamoDB which then streams data via Lambda to Amazon OpenSearch Service which you could search for all posts by different fields. For example, since the data is in Amazon OpenSearch Service you can search either the author or content fields with free-form text, even with spaces, as follows:

```
query searchName{
    searchAuthor(name:"   Nadia   "){
        id
        title
        content
    }
}

query searchContent{
    searchContent(text:"test"){
        id
        title
        content
    }
}
```

Because the data is written directly to DynamoDB, you can still perform efficient list or item lookup operations against the table with the `allPosts{...}` and `singlePost{...}` queries. This stack uses the following example code for DynamoDB streams:

Note: This code is for example only.

```
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);
};
```

You can then use DynamoDB streams to attach this to a DynamoDB table with a primary key of `id`, and any changes to the source of DynamoDB would stream into your OpenSearch Service domain. For more information about configuring this, see the DynamoDB Streams documentation.

Using DynamoDB batch operations in AWS AppSync

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

AWS AppSync supports using Amazon DynamoDB batch operations across one or more tables in a single region. Supported operations are `BatchGetItem`, `BatchPutItem`, and `BatchDeleteItem`. By using these features in AWS AppSync, you can perform tasks such as:

- Pass a list of keys in a single query and return the results from a table
- Read records from one or more tables in a single query
- Write records in bulk to one or more tables
- Conditionally write or delete records in multiple tables that might have a relation

Using batch operations with DynamoDB in AWS AppSync is an advanced technique that takes a little extra thought and knowledge of your backend operations and table structures. Additionally, batch operations in AWS AppSync have two key differences from non-batched operations:

- The data source role must have permissions to all tables which the resolver will access.
- The table specification for a resolver is part of the mapping template.

Permissions

Like other resolvers, you need to create a data source in AWS AppSync and either create a role or use an existing one. Because batch operations require different permissions on DynamoDB tables, you need to grant the configured role permissions for read or write actions:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:us-east-1:111122223333:table/TABLENAME",
                "arn:aws:dynamodb:us-east-1:111122223333:table/TABLENAME/*"
            ]
        }
    ]
}
```

Note: Roles are tied to data sources in AWS AppSync, and resolvers on fields are invoked against a data source. Data sources configured to fetch against DynamoDB only have one table specified, to keep configuration simple. Therefore, when performing a batch operation against multiple tables in a single resolver, which is a more advanced task, you must grant the role on that data source access to any tables the resolver will interact with. This would be done in the **Resource** field in the IAM policy above. Configuration of the tables to make batch calls against is done in the resolver template, which we describe below.

Data Source

For the sake of simplicity, we'll use the same data source for all the resolvers used in this tutorial. On the **Data sources** tab, create a new DynamoDB data source and name it **BatchTutorial**. The table name can be anything because table names are specified as part of the request mapping template for batch operations. We will give the table name `empty`.

For this tutorial, any role with the following inline policy will work:

Single Table Batch

> **Warning**
>
> `BatchPutItem` and `BatchDeleteItem` are not supported when used with conflict detection and resolution. These settings must be disabled to prevent possible errors.

For this example, suppose you have a single table named **Posts** to which you want to add and remove items with batch operations. Use the following schema, noting that for the query, we'll pass in a list of IDs:

```
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
}
```

Attach a resolver to the `batchAdd( )` field with the following **Request Mapping Template**. This automatically takes each item in the GraphQL `input PostInput` type and builds a map, which is needed for the `BatchPutItem` operation:

```
#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)
    }
}
```

In this case, the **Response Mapping Template** is a simple passthrough, but the table name is appended as `..data.Posts` to the context object as follows:

```
$util.toJson($ctx.result.data.Posts)
```

Now navigate to the **Queries** page of the AWS AppSync console and run the following **batchAdd** mutation:

```
mutation add {
    batchAdd(posts:[{
            id: 1 title: "Running in the Park"},{
            id: 2 title: "Playing fetch"
        }]){
            id
            title
    }
}
```

You should see the results printed to the screen, and can independently validate through the DynamoDB console that both values wrote to the **Posts** table.

Next, attach a resolver to the `batchGet()` field with the following **Request Mapping Template**. This automatically takes each item in the GraphQL `ids:[]` type and builds a map that is needed for the `BatchGetItem` operation:

```
#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"}
                }
        }
    }
}
```

The **Response Mapping Template** is again a simple passthrough, with again the table name appended as `..data.Posts` to the context object:

```
$util.toJson($ctx.result.data.Posts)
```

Now go back to the **Queries** page of the AWS AppSync console, and run the following **batchGet Query**:

```
query get {
    batchGet(ids:[1,2,3]){
        id
        title
    }
}
```

This should return the results for the two `id` values that you added earlier. Note that a `null` value returned for the `id` with a value of `3`. This is because there was no record in your **Posts** table with that value yet. Also note that AWS AppSync returns the results in the same order as the keys passed in to the query, which is an additional feature that AWS AppSync does on your behalf. So if you switch to `batchGet(ids:[1,3,2)`, you'll see the order changed. You'll also know which `id` returned a `null` value.

Finally, attach a resolver to the `batchDelete()` field with the following **Request Mapping Template**. This automatically takes each item in the GraphQL `ids:[]` type and builds a map that is needed for the `BatchGetItem` operation:

```
#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)
    }
}
```

The **Response Mapping Template** is again a simple passthrough, with again the table name appended as `..data.Posts` to the context object:

```
$util.toJson($ctx.result.data.Posts)
```

Now go back to the **Queries** page of the AWS AppSync console, and run the following **batchDelete** mutation:

```
mutation delete {
    batchDelete(ids:[1,2]){ id }
}
```

The records with `id` 1 and 2 should now be deleted. If you re-run the `batchGet()` query from earlier, these should return `null`.

Multi-Table Batch

> **Warning**
>
> `BatchPutItem` and `BatchDeleteItem` are not supported when used with conflict detection and resolution. These settings must be disabled to prevent possible errors.

AWS AppSync also enables you to perform batch operations across tables. Let's build a more complex application. Imagine we are building a Pet Health app, where sensors report the pet location and body temperature. The sensors are battery powered and attempt to connect to the network every few minutes. When a sensor establishes connection, it sends its readings to our AWS AppSync API. Triggers then analyze the data so a dashboard can be presented to the pet owner. Let's focus on representing the interactions between the sensor and the backend data store.

As a prerequisite, let's first create two DynamoDB tables; **locationReadings** will store sensor location readings and **temperatureReadings** will store sensor temperature readings. Both tables happen to share the same primary key structure: `sensorId (String)` being the partition key, and `timestamp (String)` the sort key.

Let's use the following GraphQL schema:

```
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 - Recording Sensor Readings

Our sensors need to be able to send their readings once they connect to the internet. The GraphQL field `Mutation.recordReadings` is the API they will use to do so. Let's attach a resolver to bring our API to life.

Select **Attach** next to the `Mutation.recordReadings` field. On the next screen, pick the same `BatchTutorial` data source created at the beginning of the tutorial.

Let's add the following request mapping template:

Request Mapping Template

```
## 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)
    }
}
```

As you can see, the `BatchPutItem` operation allows us to specify multiple tables.

Let's use the following response mapping template.

Response Mapping Template

```
## 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)
```

With batch operations, there can be both errors and results returned from the invocation. In that case, we're free to do some extra error handling.

Note: The use of `$utils.appendError()` is similar to the `$util.error()`, with the major distinction that it doesn't interrupt the evaluation of the mapping template. Instead, it signals

there was an error with the field, but allows the template to be evaluated and consequently return data back to the caller. We recommend you use `$utils.appendError()` when your application needs to return partial results.

Save the resolver and navigate to the **Queries** page of the AWS AppSync console. Let's send some sensor readings!

Execute the following mutation:

```
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
    }
  }
}
```

We sent 10 sensor readings in one mutation, with readings split up across two tables. Use the DynamoDB console to validate that data shows up in both the **locationReadings** and **temperatureReadings** tables.

BatchDeleteItem - Deleting Sensor Readings

Similarly, we would also need to delete batches of sensor readings. Let's use the `Mutation.deleteReadings` GraphQL field for this purpose. Select **Attach** next to the `Mutation.recordReadings` field. On the next screen, pick the same `BatchTutorial` data source created at the beginning of the tutorial.

Let's use the following request mapping template.

Request Mapping Template

```
## 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)
    }
}
```

The response mapping template is the same as the one we used for `Mutation.recordReadings`.

Response Mapping Template

```
## 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)
```

Save the resolver and navigate to the **Queries** page of the AWS AppSync console. Now, let's delete a couple of sensor readings!

Execute the following mutation:

```
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
    }
  }
}
```

Validate through the DynamoDB console that these two readings have been deleted from the **locationReadings** and **temperatureReadings** tables.

BatchGetItem - Retrieve Readings

Another common operation for our Pet Health app would be to retrieve the readings for a sensor at a specific point in time. Let's attach a resolver to the `Query.getReadings` GraphQL field on

our schema. Select **Attach**, and on the next screen pick the same `BatchTutorial` data source created at the beginning of the tutorial.

Let's add the following request mapping template.

Request Mapping Template

```
## 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
        }
    }
}
```

Note that we are now using the **BatchGetItem** operation.

Our response mapping template is going to be a little different because we chose to return a `SensorReading` list. Let's map the invocation result to the desired shape.

Response Mapping Template

```
## 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)
```

Save the resolver and navigate to the **Queries** page of the AWS AppSync console. Now, let's retrieve sensor readings!

Execute the following query:

```
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
    }
  }
}
```

We have successfully demonstrated the use of DynamoDB batch operations using AWS AppSync.

Error Handling

In AWS AppSync, data source operations can sometimes return partial results. Partial results is the term we will use to denote when the output of an operation is comprised of some data and an error. Because error handling is inherently application specific, AWS AppSync gives you the opportunity to handle errors in the response mapping template. The resolver invocation error, if present, is available from the context as `$ctx.error`. Invocation errors always include a message and a type, accessible as properties `$ctx.error.message` and `$ctx.error.type`. During the response mapping template invocation, you can handle partial results in three ways:

1. swallow the invocation error by just returning data
2. raise an error (using `$util.error(...)`) by stopping the response mapping template evaluation, which won't return any data.
3. append an error (using `$util.appendError(...)`) and also return data

Let's demonstrate each of the three points above with DynamoDB batch operations!

DynamoDB Batch operations

With DynamoDB batch operations, it is possible that a batch partially completes. That is, it is possible that some of the requested items or keys are left unprocessed. If AWS AppSync is unable to complete a batch, unprocessed items and an invocation error will be set on the context.

We will implement error handling using the `Query.getReadings` field configuration from the `BatchGetItem` operation from the previous section of this tutorial. This time, let's pretend that while executing the `Query.getReadings` field, the `temperatureReadings` DynamoDB table ran out of provisioned throughput. DynamoDB raised a **ProvisionedThroughputExceededException** at the second attempt by AWS AppSync to process the remaining elements in the batch.

The following JSON represents the serialized context after the DynamoDB batch invocation but before the response mapping template was evaluated.

```
{
  "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": []
}
```

A few things to note on the context:

- the invocation error has been set on the context at `$ctx.error` by AWS AppSync, and the error type has been set to **DynamoDB:ProvisionedThroughputExceededException**.
- results are mapped per table under `$ctx.result.data`, even though an error is present
- keys that were left unprocessed are available at `$ctx.result.data.unprocessedKeys`. Here, AWS AppSync was unable to retrieve the item with key (sensorId:1, timestamp:2018-02-01T17:21:05.000+08:00) because of insufficient table throughput.

Note: For `BatchPutItem`, it is `$ctx.result.data.unprocessedItems`. For `BatchDeleteItem`, it is `$ctx.result.data.unprocessedKeys`.

Let's handle this error in three different ways.

1. Swallowing the invocation error

Returning data without handling the invocation error effectively swallows the error, making the result for the given GraphQL field always successful.

The response mapping template we write is familiar and only focuses on the result data.

Response mapping template:

```
$util.toJson($ctx.result.data)
```

GraphQL response:

```
{
  "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
      }
    ]
  }
}
```

No errors will be added to the error response as only data was acted on.

2. Raising an error to abort the template execution

When partial failures should be treated as complete failures from the client's perspective, you can abort the template execution to prevent returning data. The `$util.error(...)` utility method achieves exactly this behavior.

Response mapping template:

```
## 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 response:

```
{
  "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. (...)"
    }
  ]
}
```

Even though some results might have been returned from the DynamoDB batch operation, we chose to raise an error such that the `getReadings` GraphQL field is null and the error has been added to the GraphQL response *errors* block.

3. Appending an error to return both data and errors

In certain cases, to provide a better user experience, applications can return partial results and notify their clients of the unprocessed items. The clients can decide to either implement a retry or translate the error back to the end user. The `$util.appendError(...)` is the utility method that enables this behavior by letting the application designer append errors on the context without

interfering with the evaluation of the template. After evaluating the template, AWS AppSync will process any context errors by appending them to the errors block of the GraphQL response.

Response mapping template:

```
#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)
```

We forwarded both the invocation error and unprocessedKeys element inside the errors block of the GraphQL response. The `getReadings` field also return partial data from the **locationReadings** table as you can see in the response below.

GraphQL response:

```
{
  "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. (...)"
    }
  ]
}
```

Performing DynamoDB transactions in AWS AppSync

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

AWS AppSync supports using Amazon DynamoDB transaction operations across one or more tables in a single region. Supported operations are `TransactGetItems` and `TransactWriteItems`. By using these features in AWS AppSync, you can perform tasks such as:

- Pass a list of keys in a single query and return the results from a table
- Read records from one or more tables in a single query
- Write records in transaction to one or more tables in an all-or-nothing way
- Execute transactions when some conditions are satisfied

Permissions

Like other resolvers, you need to create a data source in AWS AppSync and either create a role or use an existing one. Because transaction operations require different permissions on DynamoDB tables, you need to grant the configured role permissions for read or write actions:

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-1:111122223333:table/TABLENAME",
                "arn:aws:dynamodb:us-east-1:111122223333:table/TABLENAME/*"
            ]
        }
    ]
}
```

Note: Roles are tied to data sources in AWS AppSync, and resolvers on fields are invoked against a data source. Data sources configured to fetch against DynamoDB only have one table specified, to keep configuration simple. Therefore, when performing a transaction operation against multiple tables in a single resolver, which is a more advanced task, you must grant the role on that data source access to any tables the resolver will interact with. This would be done in the **Resource** field in the IAM policy above. Configuration of the transaction calls against the tables is done in the resolver template, which we describe below.

Data Source

For the sake of simplicity, we'll use the same data source for all the resolvers used in this tutorial. On the **Data sources** tab, create a new DynamoDB data source and name it **TransactTutorial**. The table name can be anything because table names are specified as part of the request mapping template for transaction operations. We will give the table name `empty`.

We'll have two tables called **savingAccounts** and **checkingAccounts**, both with `accountNumber` as partition key, and a **transactionHistory** table with `transactionId` as partition key.

For this tutorial, any role with the following inline policy will work. Replace `region` and `accountId` with your region and account ID:

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-1:111122223333:table/savingAccounts",
                "arn:aws:dynamodb:us-east-1:111122223333:table/savingAccounts/*",
                "arn:aws:dynamodb:us-east-1:111122223333:table/checkingAccounts",
                "arn:aws:dynamodb:us-east-1:111122223333:table/checkingAccounts/
*",
                "arn:aws:dynamodb:us-east-1:111122223333:table/
transactionHistory",
                "arn:aws:dynamodb:us-east-1:111122223333:table/
transactionHistory/*"
            ]
        }
    ]
}
```

Transactions

For this example, the context is a classic banking transaction, where we'll use `TransactWriteItems` to:

- Transfer money from saving accounts to checking accounts
- Generate new transaction records for each transaction

And then we'll use `TransactGetItems` to retrieve details from saving accounts and checking accounts.

> **Warning**
>
> `TransactWriteItems` is not supported when used with conflict detection and resolution. These settings must be disabled to prevent possible errors.

We define our GraphQL schema as follows:

```
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 - Populate Accounts

In order to transfer money between accounts, we need to populate the table with the details. We'll use the GraphQL operation `Mutation.populateAccounts` to do so.

In the Schema section, click on **Attach** next to the `Mutation.populateAccounts` operation. Go to VTL Unit Resolvers, then choose the same `TransactTutorial` data source.

Now use the following request mapping template:

Request Mapping Template

```
#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)
}
```

And the following response mapping template:

Response Mapping Template

```
#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)
```

Save the resolver and navigate to the **Queries** section of the AWS AppSync console to populate the accounts.

Execute the following mutation:

```
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
    }
  }
}
```

We populated 3 saving accounts and 3 checking accounts in one mutation.

Use the DynamoDB console to validate that data shows up in both the **savingAccounts** and **checkingAccounts** tables.

TransactWriteItems - Transfer Money

Attach a resolver to the `transferMoney` mutation with the following **Request Mapping Template**. Note the values of `amounts`, `savingAccountNumbers`, and `checkingAccountNumbers` are the same.

```
#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)
}
```

We will have 3 banking transactions in a single `TransactWriteItems` operation. Use the following **Response Mapping Template**:

```
#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)
```

Now navigate to the **Queries** section of the AWS AppSync console and execute the **transferMoney** mutation as follows:

```
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
    }
  }
}
```

We sent 2 banking transactions in one mutation. Use the DynamoDB console to validate that data shows up in the **savingAccounts**, **checkingAccounts**, and **transactionHistory** tables.

TransactGetItems - Retrieve Accounts

In order to retrieve the details from saving accounts and checking accounts in a single transactional request we'll attach a resolver to the `Query.getAccounts` GraphQL operation on our schema. Select **Attach**, go to VTL Unit Resolvers, then on the next screen, pick the same `TransactTutorial` data source created at the beginning of the tutorial. Configure the templates as follows:

Request Mapping Template

```
#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)
}
```

Response Mapping Template

```
#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)
```

Save the resolver and navigate to the **Queries** sections of the AWS AppSync console. In order to retrieve the saving accounts and checing accounts, execute the following query:

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
      username
      balance
    }
  }
}
```

We have successfully demonstrated the use of DynamoDB transactions using AWS AppSync.

Using HTTP resolvers in AWS AppSync

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

AWS AppSync enables you to use supported data sources (that is, AWS Lambda, Amazon DynamoDB, Amazon OpenSearch Service, or Amazon Aurora) to perform various operations, in addition to any arbitrary HTTP endpoints to resolve GraphQL fields. After your HTTP endpoints are available, you can connect to them using a data source. Then, you can configure a resolver in the schema to perform GraphQL operations such as queries, mutations, and subscriptions. This tutorial walks you through some common examples.

In this tutorial you use a REST API (created using Amazon API Gateway and Lambda) with an AWS AppSync GraphQL endpoint.

One-Click Setup

If you want to automatically set up a GraphQL endpoint in AWS AppSync with an HTTP endpoint configured (using Amazon API Gateway and Lambda), you can use the following AWS CloudFormation template :

Creating a REST API

You can use the following AWS CloudFormation template to set up a REST endpoint that works for this tutorial:

The AWS CloudFormation stack performs the following steps:

1. Sets up a Lambda function that contains your business logic for your microservice.
2. Sets up an API Gateway REST API with the following endpoint/method/content type combination:

API Resource Path	HTTP Method	Supported Content Type
/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

Creating Your GraphQL API

To create the GraphQL API in AWS AppSync:

- Open the AWS AppSync console and choose **Create API**.
- For the API name, type `UserData`.
- Choose **Custom schema**.
- Choose **Create**.

The AWS AppSync console creates a new GraphQL API for you using the API key authentication mode. You can use the console to set up the rest of the GraphQL API and run queries on it for the remainder of this tutorial.

Creating a GraphQL Schema

Now that you have a GraphQL API, let's create a GraphQL schema. From the schema editor in the AWS AppSync console, make sure you schema matches the following schema:

```
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
}
```

Configure Your HTTP Data Source

To configure your HTTP data source, do the following:

- On the **DataSources** tab, choose **New**, and then type a friendly name for the data source (for example, HTTP).
- In **Data source type**, choose **HTTP**.
- Set the endpoint to the API Gateway endpoint that is created. Make sure that you don't include the stage name as part of the endpoint.

Note: At this time only public endpoints are supported by AWS AppSync.

Note: For more information about the certifying authorities that are recognized by the AWS AppSync service, see Certificate Authorities (CA) Recognized by AWS AppSync for HTTPS Endpoints.

Configuring Resolvers

In this step, you connect the http data source to the **getUser** query.

To set up the resolver:

- Choose the **Schema** tab.
- In the **Data types** pane on the right under the **Query** type, find the **getUser** field and choose **Attach**.
- In **Data source name**, choose **HTTP**.
- In **Configure the request mapping template**, paste the following code:

```
{
    "version": "2018-05-29",
    "method": "GET",
    "params": {
        "headers": {
```

```
            "Content-Type": "application/json"
        }
    },
    "resourcePath": $util.toJson("/v1/users/${ctx.args.id}")
}
```

- In **Configure the response mapping template**, paste the following code:

```
## 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
```

- Choose the **Query** tab, and then run the following query:

```
query GetUser{
    getUser(id:1){
        id
        username
    }
}
```

This should return the following response:

```
{
    "data": {
        "getUser": {
            "id": "1",
            "username": "nadia"
        }
    }
}
```

- Choose the **Schema** tab.
- In the **Data types** pane on the right under **Mutation**, find the **addUser** field and choose **Attach**.

- In **Data source name**, choose **HTTP**.
- In **Configure the request mapping template**, paste the following code:

```
{
    "version": "2018-05-29",
    "method": "POST",
    "resourcePath": "/v1/users",
    "params":{
      "headers":{
        "Content-Type": "application/json",
      },
      "body": $util.toJson($ctx.args.userInput)
    }
}
```

- In **Configure the response mapping template**, paste the following code:

```
## 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
```

- Choose the **Query** tab, and then run the following query:

```
mutation addUser{
    addUser(userInput:{
        id:"2",
        username:"shaggy"
    }){
        id
```

```
        username
    }
}
```

This should return the following response:

```
{
    "data": {
        "getUser": {
        "id": "2",
        "username": "shaggy"
        }
    }
}
```

Invoking AWS Services

You can use HTTP resolvers to set up a GraphQL API interface for AWS services. HTTP requests to AWS must be signed with the Signature Version 4 process so that AWS can identify who sent them. AWS AppSync calculates the signature on your behalf when you associate an IAM role with the HTTP data source.

You provide two additional components to invoke AWS services with HTTP resolvers:

- An IAM role with permissions to call the AWS service APIs
- Signing configuration in the data source

For example, if you want to call the ListGraphqlApis operation with HTTP resolvers, you first create an IAM role that AWS AppSync assumes with the following policy attached:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "appsync:ListGraphqlApis"
            ],
            "Effect": "Allow",
```

```
            "Resource": "*"
        }
    ]
}
```

Next, create the HTTP data source for AWS AppSync. In this example, you call AWS AppSync in the US West (Oregon) Region. Set up the following HTTP configuration in a file named `http.json`, which includes the signing region and service name:

```
{
    "endpoint": "https://appsync.us-west-2.amazonaws.com/",
    "authorizationConfig": {
        "authorizationType": "AWS_IAM",
        "awsIamConfig": {
            "signingRegion": "us-west-2",
            "signingServiceName": "appsync"
        }
    }
}
```

Then, use the AWS CLI to create the data source with an associated role as follows:

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

When you attach a resolver to the field in the schema, use the following request mapping template to call AWS AppSync:

```
{
    "version": "2018-05-29",
    "method": "GET",
    "resourcePath": "/v1/apis"
}
```

When you run a GraphQL query for this data source, AWS AppSync signs the request using the role you provided and includes the signature in the request. The query returns a list of AWS AppSync GraphQL APIs in your account in that AWS Region.

Using Aurora Serverless v2 with AWS AppSync

Connect your GraphQL API to Aurora Serverless databases using AWS AppSync. This integration lets you execute SQL statements through GraphQL queries, mutations, and subscriptions - giving you a flexible way to interact with your relational data.

> **Note**
>
> This tutorial uses the `US-EAST-1` Region.

Benefits

- Seamless integration between GraphQL and relational databases
- Ability to perform SQL operations through GraphQL interfaces
- Serverless scalability with Aurora Serverless v2
- Secure data access through AWS Secrets Manager
- Protection against SQL injection through input sanitization
- Flexible query capabilities including filtering and range operations

Common Use Cases

- Building scalable applications with relational data requirements
- Creating APIs that need both GraphQL flexibility and SQL database capabilities
- Managing data operations through GraphQL mutations and queries
- Implementing secure database access patterns

In this tutorial, you will learn the following.

- Set up an Aurora Serverless v2 cluster
- Enable Data API functionality
- Create and configure database structures
- Define GraphQL schemas for database operations
- Implement resolvers for queries and mutations
- Secure your data access through proper input sanitization

- Execute various database operations through GraphQL interfaces

Topics

Setting up your database cluster

Before adding an Amazon RDS data source to AWS AppSync, you must first enable a Data API on an Aurora Serverless v2 cluster and **configure a secret** using *AWS Secrets Manager*. You can create an Aurora Serverless v2 cluster using the 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
```

This will return an ARN for the cluster.

After creating the cluster, you must add an Aurora Serverless v2 instance using the following command.

```
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

These endpoints take time to activate. You can check their status in the Amazon RDS console in the **Connectivity & security** tab for the cluster. You can also check the status of your cluster with the following AWS CLI command.

```
aws rds describe-db-clusters \
    --db-cluster-identifier appsync-tutorial \
    --query "DBClusters[0].Status"
```

You can create a *Secret* using the AWS Secrets Manager Console or the AWS CLI with an input file such as the following using the `USERNAME` and `COMPLEX_PASSWORD` from the previous step.

```
{
    "username": "USERNAME",
    "password": "COMPLEX_PASSWORD"
}
```

Pass this as a parameter to the AWS CLI:

```
aws secretsmanager create-secret --name HttpRDSSecret --secret-string file://creds.json
 --region us-east-1
```

This will return an ARN for the secret.

Note the ARN of your Aurora Serverless cluster and Secret for later use in the AppSync console when creating a data source.

Enable Data API

You can enable the Data API on your cluster by following the instructions in the RDS documentation. The Data API must be enabled before adding as an AppSync data source.

Create database and table

Once you have enabled your Data API you can ensure it works with the `aws rds-data execute-statement` command in the AWS CLI. This will ensure that your Aurora Serverless cluster is

configured correctly before adding it to your AppSync API. First create a database called *TESTDB* with the `--sql` parameter like so:

```
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"
```

If this runs without error, add a table with the *create table* command:

```
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"
```

If everything has run without issue you can move forward to adding the cluster as a data source in your AppSync API.

GraphQL schema

Now that your Aurora Serverless Data API is up and running with a table, we will create a GraphQL schema and attach resolvers for performing mutations and subscriptions. Create a new API in the AWS AppSync console and navigate to the **Schema** page, and enter the following:

```
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
}
```

Save your schema and navigate to the **Data Sources** page and create a new data source. Select **Relational database** for the Data source type, and provide a friendly name. Use the database name that you created in the last step, as well as the **Cluster ARN** that you created it in. For the **Role** you can either have AppSync create a new role or create one with a policy similar to the below:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": [
                "rds-data:BatchExecuteStatement",
                "rds-data:BeginTransaction",
                "rds-data:CommitTransaction",
                "rds-data:ExecuteStatement",
                "rds-data:RollbackTransaction"
            ],
            "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:*"
            ]
        }
    ]
}
```

Note there are two **Statements** in this policy which you are granting role access. The first **Resource** is your Aurora Serverless cluster and the second is your AWS Secrets Manager ARN. You will need to provide **BOTH** ARNs in the AppSync data source configuration before clicking **Create**.

Pass this as a parameter to the AWS CLI.

```
aws secretsmanager create-secret \
  --name HttpRDSSecret \
  --secret-string file://creds.json \
  --region us-east-1
```

This will return an ARN for the secret. Take note of the ARN of your Aurora Serverless cluster and Secret for later when creating a data source in the AWS AppSync console.

Build Your Database Structure

Once you have enabled your Data API you can ensure it works with the `aws rds-data execute-statement` command in the AWS CLI. This will ensure that your Aurora Serverless v2 cluster is configured correctly before adding it to your AWS AppSync API. First, create a database called *TESTDB* with the `--sql` parameter as follows.

```
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"
```

If this runs without errors, add a table with the following *create table* command.

```
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"
```

Design Your API Interface

After Aurora Serverless v2 Data API is up and running with a table, create a GraphQL schema and attach resolvers for performing mutations and subscriptions. Create a new API in the AWS AppSync console and navigate to the **Schema** page in the console, and enter the following.

```
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
}
```

Save your schema and navigate to the **Data Sources** page and create a new data source. Choose **Relational database** for the **Data source** type, and provide a friendly name. Use the database name that you created in the last step, as well as the **Cluster ARN** that you created it in. For the **Role** you can either have AWS AppSync create a new role or create one with a policy similar to the following.

JSON

```
{
        "Version": "2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Action": [
                    "rds-data:BatchExecuteStatement",
                    "rds-data:BeginTransaction",
                    "rds-data:CommitTransaction",
                    "rds-data:ExecuteStatement",
                    "rds-data:RollbackTransaction"
                ],
                "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:*"
                ]
            }
        ]
    }
```

Note there are two **Statements** in this policy which you are granting role access. The first **Resource** is your Aurora Serverless v2 cluster and the second is your AWS Secrets Manager ARN. You will need to provide **BOTH** ARNs in the AWS AppSync data source configuration before clicking **Create**.

Connect Your API to Database Operations

Now that we have a valid GraphQL schema and an RDS data source, you can attach resolvers to the GraphQL fields to your schema. Our API will offer the following capabilities:

1. create a pet using the *Mutation.createPet* field
2. update a pet using the *Mutation.updatePet* field
3. delete a pet using the *Mutation.deletePet* field
4. get a single using via the *Query.getPet* field
5. list all using the *Query.listPets* field
6. list pets in a price range using the *Query.listPetsByPriceRange* field

Mutation.createPet

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `createPet(input: CreatePetInput!): Pet`. Choose your RDS data source. In the **request mapping template** section, add the following template:

```
#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)
    }
}
```

The system executes SQL statements sequentially, based on the order in the **statements** array. The results will come back in the same order. Since this is a mutation, you will run a *select* statement after the *insert* to retrieve the committed values in order to populate the GraphQL response mapping template.

In the **response mapping template** section, add the following template:

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

Because the *statements* has two SQL queries, we need to specify the second result in the matrix that comes back from the database with: `$utils.rds.toJsonString($ctx.result))[1][0])`.

Mutation.updatePet

From the schema editor in the AWS AppSync console, choose **Attach Resolver** for `updatePet(input: UpdatePetInput!): Pet`. Choose your **RDS data source**. In the **request mapping template** section, add the following template.

```
{
"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)
    }
}
```

In the **response mapping template** section, add the following template.

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

Mutation.deletePet

From the schema editor in the AWS AppSync console, choose **Attach Resolver** for `deletePet(input: DeletePetInput!): Pet`. Choose your **RDS data source**. In the **request mapping template** section, add the following template.

```
{
"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"
    }
}
```

In the **response mapping template** section, add the following template.

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.getPet

Now that the mutations are created for your schema, connect the three queries to showcase how to get individual items, lists, and apply SQL filtering. From the **schema editor** in the AWS AppSync console, choose **Attach Resolver** for `getPet(id: ID!): Pet`. Choose your **RDS data source**. In the **request mapping template** section, add the following template.

```
{
"version": "2018-05-29",
        "statements": [
            $util.toJson("select * from Pets WHERE id=:ID")
    ],
    "variableMap": {
        ":ID": "$ctx.args.id"
    }
}
```

In the **response mapping template** section, add the following template:

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.listPets

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `getPet(id: ID!): Pet`. Choose your **RDS data source**. In the **request mapping template** section, add the following template.

```
{
    "version": "2018-05-29",
    "statements": [
        "select * from Pets"
```

```
    ]
}
```

In the **response mapping template** section, add the following template.

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

Query.listPetsByPriceRange

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `getPet(id: ID!): Pet`. Choose your **RDS data source**. In the **request mapping template** section, add the following template.

```
{
    "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)
    }
}
```

In the **response mapping template** section, add the following template:

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

Modify Your Data Through the API

Now that you have configured all of your resolvers with SQL statements and connected your GraphQL API to your Serverless Aurora Data API, you can begin performing mutations and queries. In AWS AppSync console, choose the **Queries** tab and enter the following to create a Pet:

```
mutation add {
    createPet(input : { type:fish, price:10.0 }){
        id
        type
```

```
        price
    }
}
```

The response should contain the *id*, *type*, and *price* like so:

```
{
  "data": {
    "createPet": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "type": "fish",
      "price": "10.0"
    }
  }
}
```

You can modify this item by running the *updatePet* mutation:

```
mutation update {
    updatePet(input : {
        id: ID_PLACEHOLDER,
        type:bird,
        price:50.0
    }){
        id
        type
        price
    }
}
```

Note that we used the *id* which was returned from the *createPet* operation earlier. This will be a unique value for your record as the resolver leveraged `$util.autoId()`. You could delete a record in a similar manner:

```
mutation delete {
    deletePet(input : {id:ID_PLACEHOLDER}){
        id
        type
        price
    }
}
```

Create a few records with the first mutation with different values for *price* and then run some queries.

Retrieve Your Data

Still in the **Queries** tab of the console, use the following statement to list all of the records you've created.

```
query allpets {
    listPets {
        id
        type
        price
    }
}
```

Leverage the SQL *WHERE* predicate that had `where price > :MIN and price < :MAX` in our mapping template for *Query.listPetsByPriceRange* with the following GraphQL query:

```
query petsByPriceRange {
    listPetsByPriceRange(min:1, max:11) {
        id
        type
        price
    }
}
```

You should only see records with a *price* over $1 or less than $10. Finally, you can perform queries to retrieve individual records as follows:

```
query onePet {
    getPet(id:ID_PLACEHOLDER){
        id
        type
        price
    }
}
```

Secure Your Data Access

SQL injection is a security vulnerability in database applications. It occurs when attackers insert malicious SQL code through user input fields. This can allow unauthorized access to database data. We recommend that you carefully validate and sanitize all user inputs before processing using `variableMap` for protection against SQL injection attacks. If variable maps are not used, you are responsible for sanitizing the arguments of their GraphQL operations. One way to do this is to provide input specific validation steps in the request mapping template before execution of a SQL statement against your Data API. Let's see how we can modify the request mapping template of the `listPetsByPriceRange` example. Instead of relying solely on the user input you can do the following:

```
#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)
    }
}
```

Another way to protect against rogue input when executing resolvers against your Data API is to use prepared statements together with stored procedure and parameterized inputs. For example, in the resolver for `listPets` define the following procedure that executes the *select* as a prepared statement:

```
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
```

Create this in your Aurora Serverless v2 Instance.

```
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"
```

The resulting resolver code for listPets is simplified since we now simply call the stored procedure. At a minimum, any string input should have single quotes [escaped].

```
#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("'", "''"))
    }
}
```

Using escape strings

Use single quotes to mark the start and end of string literals in an SQL statement e.g.. `'some string value'`. To allow string values with one or more single quote characters (`'`) to be used within a string, each must be replaced with two single quotes (`''`). For example, if the input string is `Nadia's` dog, you would escape it for the SQL statement like

```
update Pets set type='Nadia''s dog' WHERE id='1'
```

Using pipeline resolvers in AWS AppSync

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

AWS AppSync provides a simple way to wire a GraphQL field to a single data source through unit resolvers. However, executing a single operation might not be enough. Pipeline resolvers offer the ability to serially execute operations against data sources. Create functions in your API and attach them to a pipeline resolver. Each function execution result is piped to the next until no function is left to execute. With pipeline resolvers you can now build more complex workflows directly in AWS AppSync. In this tutorial, you build a simple pictures viewing app, where users can post and view pictures posted by their friends.

One-Click Setup

If you want to automatically set up the GraphQL endpoint in AWS AppSync with all the resolvers configured and the necessary AWS resources, you can use the following AWS CloudFormation template :

This stack creates the following resources in your account:

- IAM Role for AWS AppSync to access the resources in your account
- 2 DynamoDB tables
- 1 Amazon Cognito user pool
- 2 Amazon Cognito user pool groups
- 3 Amazon Cognito user pool users
- 1 AWS AppSync API

At the end of the AWS CloudFormation stack creation process you receive one email for each of the three Amazon Cognito users that were created. Each email contains a temporary password that you use to log in as an Amazon Cognito user to the AWS AppSync console. Save the passwords for the remainder of the tutorial.

Manual Setup

If you prefer to manually go through a step-by-step process through the AWS AppSync console, follow the setup process below.

Setting Up Your Non AWS AppSync Resources

The API communicates with two DynamoDB tables: a **pictures** table that stores pictures and a **friends** table that stores relationships between users. The API is configured to use Amazon Cognito user pool as authentication type. The following AWS CloudFormation stack sets up these resources in the account.

At the end of the AWS CloudFormation stack creation process you receive one email for each of the three Amazon Cognito users that were created. Each email contains a temporary password that you use to log in as an Amazon Cognito user to the AWS AppSync console. Save the passwords for the remainder of the tutorial.

Creating Your GraphQL API

To create the GraphQL API in AWS AppSync:

1. Open the AWS AppSync console and choose **Build From Scratch** and choose **Start**.
2. Set the name of the API to `AppSyncTutorial-PicturesViewer`.
3. Choose **Create**.

The AWS AppSync console creates a new GraphQL API for you using the API key authentication mode. You can use the console to set up the rest of the GraphQL API and run queries against it for the rest of this tutorial.

Configuring The GraphQL API

You need to configure the AWS AppSync API with the Amazon Cognito user pool that you just created.

1. Choose the **Settings** tab.
2. Under the **Authorization Type** section, choose *Amazon Cognito User Pool*.
3. Under **User Pool Configuration**, choose **US-WEST-2** for the *AWS Region*.
4. Choose the **AppSyncTutorial-UserPool** user pool.
5. Choose **DENY** as *Default Action*.
6. Leave the **AppId client regex** field blank.
7. Choose **Save**.

The API is now set up to use Amazon Cognito user pool as its authorization type.

Configuring Data Sources for the DynamoDB Tables

After the DynamoDB tables have been created, navigate to your AWS AppSync GraphQL API in the console and choose the **Data Sources** tab. Now, you're going to create a datasource in AWS AppSync for each of the DynamoDB tables that you just created.

1. Choose the **Data source** tab.
2. Choose **New** to create a new data source.
3. For the data source name, enter `PicturesDynamoDBTable`.
4. For data source type, choose **Amazon DynamoDB table**.
5. For region, choose **US-WEST-2**.
6. From the list of tables, choose the **AppSyncTutorial-Pictures**DynamoDB table.
7. In the **Create or use an existing role** section, choose **Existing role**.
8. Choose the role that was just created from the CloudFormation template. If you did not change the *ResourceNamePrefix*, the name of the role should be **AppSyncTutorial-DynamoDBRole**.
9. Choose **Create**.

Repeat the same process for the **friends** table, the name of the DynamoDB table should be **AppSyncTutorial-Friends** if you did not change the *ResourceNamePrefix* parameter at the time of creating the CloudFormation stack.

Creating the GraphQL Schema

Now that the data sources are connected to your DynamoDB tables, let's create a GraphQL schema. From the schema editor in the AWS AppSync console, make sure your schema matches the following schema:

```
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!
}
```

Choose **Save Schema** to save your schema.

Some of the schema fields have been annotated with the *@aws_auth* directive. Since the API default action configuration is set to *DENY*, the API rejects all users that are not members of the groups mentioned inside the *@aws_auth* directive. For more information about how to secure your API, you can read the Security page. In this case, only admin users have access to the *Mutation.createPicture* and *Mutation.createFriendship* fields, while users that are members of either

Admins or *Viewers* groups can access the *Query.getPicturesByOwner* field. All other users don't have access.

Configuring Resolvers

Now that you have a valid GraphQL schema and two data sources, you can attach resolvers to the GraphQL fields on the schema. The API offers the following capabilities:

- Create a picture via the *Mutation.createPicture* field
- Create friendship via the *Mutation.createFriendship* field
- Retrieve a picture via the *Query.getPicture* field

Mutation.createPicture

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `createPicture(input: CreatePictureInput!): Picture!`. Choose the DynamoDB*PicturesDynamoDBTable* data source. In the **request mapping template** section, add the following template:

```
#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)
}
```

In the **response mapping template** section, add the following template:

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

The create picture functionality is done. You are saving a picture in the **Pictures** table, using a randomly generated UUID as id of the picture, and using the Cognito username as owner of the picture.

Mutation.createFriendship

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `createFriendship(id: ID!, target: ID!): Boolean`. Choose the DynamoDB**FriendsDynamoDBTable** data source. In the **request mapping template** section, add the following template:

```
#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)
    }
}
```

Important: In the **BatchPutItem** request template, the exact name of the DynamoDB table should be present. The default table name is *AppSyncTutorial-Friends*. If you are using the wrong table name, you get an error when AppSync tries to assume the provided role.

For the sake of simplicity in this tutorial, proceed as if the friendship request has been approved and save the relationship entry directly into the **AppSyncTutorialFriends** table.

Effectively, you're storing two items for each friendship as the relationship is bi-directional. For more details about Amazon DynamoDB best practices to represent many-to-many relationships, see DynamoDB Best Practices .

In the **response mapping template** section, add the following template:

```
#if($ctx.error)
```

```
    $util.error($ctx.error.message, $ctx.error.type)
#end
true
```

Note: Make sure your request template contains the right table name. The default name is *AppSyncTutorial-Friends*, but your table name might differ if you changed the CloudFormation **ResourceNamePrefix** parameter.

Query.getPicturesByOwner

Now that you have friendships and pictures, you need to provide the ability for users to view their friends' pictures. To satisfy this requirement, you need to first check that the requester is friend with the owner, and finally query for the pictures.

Because this functionality requires two data source operations, you're going to create two functions. The first function, **isFriend**, checks whether the requester and the owner are friends. The second function, **getPicturesByOwner**, retrieves the requested pictures given an owner ID. Let's look at the execution flow below for the proposed resolver on the *Query.getPicturesByOwner* field:

1. Before mapping template: Prepare the context and field input arguments.
2. isFriend function: Checks whether the requester is the owner of the picture. If not, it checks whether the requester and owner users are friends by doing a DynamoDB GetItem operation on the friends table.
3. getPicturesByOwner function: Retrieves pictures from the Pictures table using a DynamoDB Query operation on the *owner-index* Global Secondary Index.
4. After mapping template: Maps picture result so DynamoDB attributes map correctly to the expected GraphQL type fields.

Let's first create the functions.

isFriend Function

1. Choose the **Functions** tab.
2. Choose **Create Function** to create a function.
3. For the data source name, enter `FriendsDynamoDBTable`.
4. For the function name, enter *isFriend*.
5. Inside the request mapping template text area, paste the following template:

```
#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. Inside the response mapping template text area, paste the following template:

```
#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. Choose **Create Function**.

Result: You've created the **isFriend** function.

getPicturesByOwner function

1. Choose the **Functions** tab.
2. Choose **Create Function** to create a function.

3. For the data source name, enter `PicturesDynamoDBTable`.
4. For the function name, enter `getPicturesByOwner`.
5. Inside the request mapping template text area, paste the following template:

```
{
    "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. Inside the response mapping template text area, paste the following template:

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end

$util.toJson($ctx.result)
```

7. Choose **Create Function**.

Result: You've created the **getPicturesByOwner** function. Now that the functions have been created, attach a pipeline resolver to the *Query.getPicturesByOwner* field.

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `Query.getPicturesByOwner(id: ID!): [Picture]`. On the following page, choose the **Convert to pipeline resolver** link that appears underneath the data source drop-down list. Use the following for the before mapping template:

```
#set($result = { "owner": $ctx.args.id, "callerId": $ctx.identity.username })
```

```
$util.toJson($result)
```

In the **after mapping template** section, use the following:

```
#foreach($picture in $ctx.result.items)
    ## prepend "src://" to picture.src property
    #set($picture['src'] = "src://${picture['src']}")
#end
$util.toJson($ctx.result.items)
```

Choose **Create Resolver**. You have successfully attached your first pipeline resolver. On the same page, add the two functions you created previously. In the functions section, choose **Add A Function** and then choose or type the name of the first function, **isFriend**. Add the second function by following the same process for the **getPicturesByOwner** function. Make sure the **isFriend** function appears first in the list followed by the **getPicturesByOwner** function. You can use the up and down arrows to rearrange to order of execution of the functions in the pipeline.

Now that the pipeline resolver is created and you've attached the functions, let's test the newly created GraphQL API.

Testing Your GraphQL API

First, you need to populate pictures and friendships by executing a few mutations using the admin user you created. On the left side of the AWS AppSync console, choose the **Queries** tab.

createPicture Mutation

1. In AWS AppSync console, choose the **Queries** tab.
2. Choose **Login With User Pools**.
3. On the modal, enter the Cognito Sample Client ID that was created by the CloudFormation stack for example, 37solo6mmhh7k4v63cqdfgdg5d).
4. Enter the user name you passed as parameter to the CloudFormation stack. Default is **nadia**.
5. Use the temporary password that was sent to the email you provided as parameter to the CloudFormation stack (for example, *UserPoolUserEmail*).
6. Choose Login. You should now see the button renamed to **Logout nadia**, or whatever user name you chose when creating the CloudFormation stack (that is, *UserPoolUsername*).

Let's send a few *createPicture* mutations to populate the pictures table. Execute the following GraphQL query inside the console:

```
mutation {
  createPicture(input:{
    owner: "nadia"
    src: "nadia.jpg"
  }) {
    id
    owner
    src
  }
}
```

The response should look like below:

```
{
  "data": {
    "createPicture": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "owner": "nadia",
      "src": "nadia.jpg"
    }
  }
}
```

Let's add a few more pictures:

```
mutation {
  createPicture(input:{
    owner: "shaggy"
    src: "shaggy.jpg"
  }) {
    id
    owner
    src
  }
}
```

```
mutation {
  createPicture(input:{
```

```
    owner: "rex"
    src: "rex.jpg"
  }) {
    id
    owner
    src
  }
}
```

You've added three pictures using **nadia** as the admin user.

createFriendship Mutation

Let's add a friendship entry. Execute the following mutations in the console.

Note: You must still be logged in as the admin user (the default admin user is **nadia**).

```
mutation {
  createFriendship(id: "nadia", target: "shaggy")
}
```

The response should look like:

```
{
  "data": {
    "createFriendship": true
  }
}
```

nadia and **shaggy** are friends. **rex** is not friends with anybody.

getPicturesByOwner Query

For this step, log in as the **nadia** user using Cognito User Pools, using the credentials set up in the beginning of this tutorial. As **nadia**, retrieve the pictures owned by **shaggy**.

```
query {
    getPicturesByOwner(id: "shaggy") {
        id
        owner
        src
    }
```

```
}
```

Since **nadia** and **shaggy** are friends, the query should return the corresponding picture.

```
{
  "data": {
    "getPicturesByOwner": [
      {
        "id": "05a16fba-cc29-41ee-a8d5-4e791f4f1079",
        "owner": "shaggy",
        "src": "src://shaggy.jpg"
      }
    ]
  }
}
```

Similarly, if **nadia** attempts to retrieve her own pictures, it also succeeds. The pipeline resolver has been optimized to avoid running the **isFriend** GetItem operation in that case. Try the following query:

```
query {
    getPicturesByOwner(id: "nadia") {
        id
        owner
        src
    }
}
```

If you enable logging on your API (in the **Settings** pane), set the debug level to **ALL**, and run the same query again, it returns logs for the field execution. By looking at the logs, you can determine whether the **isFriend** function returned early at the **Request Mapping Template** stage:

```
{
  "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
}
```

The *earlyReturnedValue* key represents the data that was returned by the *#return* directive.

Finally, even though **rex** is a member of the **Viewers** Cognito UserPool Group, and because **rex** isn't friends with anybody, he won't be able to access any of the pictures owned by **shaggy** or **nadia**. If you log in as **rex** in the console and execute the following query:

```
query {
    getPicturesByOwner(id: "nadia") {
        id
        owner
        src
    }
}
```

You get the following unauthorized error:

```
{
  "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"
    }
  ]
}
```

You have successfully implemented complex authorization using pipeline resolvers.

Using Delta Sync operations on versioned data sources in AWS AppSync

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

Client applications in AWS AppSync store data by caching GraphQL responses locally to disk in a mobile/web application. Versioned data sources and Sync operations give customers the ability to perform the sync process using a single resolver. This allows clients to hydrate their local cache with results from one base query that might have a lot of records, and then receive only the data altered since their last query (the *delta updates*). By allowing clients to perform the base hydration of the cache with an initial request and incremental updates in another, you can move the computation from your client application to the backend. This is substantially more efficient for client applications that frequently switch between online and offline states.

To implement Delta Sync, the Sync query uses the Sync operation on a versioned data source. When an AWS AppSync mutation changes an item in a versioned data source, a record of that

change will be stored in the *Delta* table as well. You can choose to use different *Delta* tables (e.g. one per type, one per domain area) for other versioned data sources or a single *Delta* table for your API. AWS AppSync recommends against using a single *Delta* table for multiple APIs to avoid the collision of primary keys.

In addition, Delta Sync clients can also receive a subscription as an argument, and then the client coordinates subscription reconnects and writes between offline to online transitions. Delta Sync performs this by automatically resuming subscriptions (including exponential backoff and retry with jitter through different network error scenarios), and storing events in a queue. The appropriate delta or base query is then run before merging any events from the queue, and finally processing subscriptions as normal.

Documentation for client configuration options, including the Amplify DataStore, is available on the Amplify Framework website. This documentation outlines how to set up versioned DynamoDB data sources and `Sync` operations to work with the Delta Sync client for optimal data access.

One-Click Setup

To automatically set up the GraphQL endpoint in AWS AppSync with all the resolvers configured and the necessary AWS resources, use this AWS CloudFormation template:

This stack creates the following resources in your account:

- 2 DynamoDB tables (Base and Delta)
- 1 AWS AppSync API with API key
- 1 IAM Role with policy for DynamoDB tables

Two tables are used to partition your sync queries into a second table that acts as a journal of missed events when the clients were offline. To keep the queries efficient on the delta table, Amazon DynamoDB TTLs are used to automatically groom the events as necessary. The TTL time is configurable for your needs on the data source (you might want this as 1hour, 1day, etc.).

Schema

To demonstrate Delta Sync, the sample application creates a *Posts* schema backed by a *Base* and *Delta* table in DynamoDB. AWS AppSync automatically writes the mutations to both tables. The

sync query pulls records from the *Base* or *Delta* table as appropriate, and a single subscription is defined to show how clients can leverage this in their reconnection logic.

```
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
}
```

The GraphQL schema is standard, but a couple things are worth calling out before moving forward. First, all of the mutations automatically first write to the *Base* table and then to the *Delta* table. The *Base* table is the central source of truth for state while the *Delta* table is your journal. If you don't pass in the `lastSync: AWSTimestamp`, the `syncPosts` query runs against the *Base* table and hydrates the cache as well as running at periodic times as a *global catchup process* for edge cases when clients are offline longer than your configured TTL time in the *Delta* table. If you do

pass in the `lastSync: AWSTimestamp`, the `syncPosts` query runs against your *Delta* table and is used by clients to retrieve changed events since they were last offline. Amplify clients automatically pass the `lastSync: AWSTimestamp` value, and persist to disk appropriately.

The *_deleted* field on *Post* is used for **DELETE** operations. When clients are offline and records are removed from the *Base* table, this attribute notifies clients performing synchronization to evict items from their local cache. In cases where clients are offline for longer periods of time and the item has been removed before the client can retrieve this value with a Delta Sync query, the global catch-up event in the base query (configurable in the client) runs and removes the item from the cache. This field is marked optional because it only returns a value when running a sync query that has deleted items present.

Mutations

For all of the mutations, AWS AppSync does a standard Create/Update/Delete operation in the *Base* table and also records the change in the *Delta* table automatically. You can reduce or extend the time to keep records by modifying the `DeltaSyncTableTTL` value on the data source. For organizations with a high velocity of data, it may make sense to keep this short. Alternatively, if your clients are offline for longer periods of time, it might be prudent to keep this longer.

Sync Queries

The *base query* is a DynamoDB Sync operation without a `lastSync` value specified. For many organizations, this works because the base query only runs on startup and at a periodic basis thereafter.

The *delta query* is a DynamoDB Sync operation with a `lastSync` value specified. The *delta query* executes whenever the client comes back online from an offline state (as long as the base query periodic time hasn't triggered to run). Clients automatically track the last time they successfully ran a query to sync data.

When a delta query is run, the query's resolver uses the `ds_pk` and `ds_sk` to query only for the records that have changed since the last time the client performed a sync. The client stores the appropriate GraphQL response.

For more information on executing Sync Queries, see the Sync Operation documentation.

Example

Let's start first by calling a `createPost` mutation to create an item:

```
mutation create {
  createPost(input: {author: "Nadia", title: "My First Post", content: "Hello World"})
 {
    id
    author
    title
    content
    _version
    _lastChangedAt
    _deleted
  }
}
```

The return value of this mutation will look as follows:

```
{
  "data": {
    "createPost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
      "author": "Nadia",
      "title": "My First Post",
      "content": "Hello World",
      "_version": 1,
      "_lastChangedAt": 1574469356331,
      "_deleted": null
    }
  }
}
```

If you examine the contents of the *Base* table, you will see a record that looks like:

```
{
  "_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"
  }
}
```

If you examine the contents of the *Delta* table, you will see a record that looks like:

```
{
  "_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"
  }
}
```

Now we can simulate a *Base* query that a client will run to hydrate its local data store using a `syncPosts` query like:

```
query baseQuery {
  syncPosts(limit: 100, lastSync: null, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
      _lastChangedAt
    }
    startedAt
    nextToken
  }
}
```

The return value of this *Base* query will look as follows:

```
{
  "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
    }
  }
}
```

We'll save the `startedAt` value later to simulate a *Delta* query, but first we need to make a change to our table. Let's use the `updatePost` mutation to modify our existing Post:

```
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
  }
}
```

The return value of this mutation will look as follows:

```
{
  "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
    }
  }
}
```

If you examine the contents of the *Base* table now, you should see the updated item:

```
{
  "_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"
  }
}
```

If you examine the contents of the *Delta* table now, you should see two records:

1. A record when the item was created
2. A record for when the item was updated.

The new item will look like:

```
{
  "_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"
  }
}
```

Now we can simulate a *Delta* query to retrieve modifications that occurred when a client was offline. We will use the `startedAt` value returned from our *Base* query to make the request:

```
query delta {
  syncPosts(limit: 100, lastSync: 1574469602238, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
    }
    startedAt
    nextToken
  }
}
```

The return value of this *Delta* query will look as follows:

```
{
  "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
    }
  }
}
```

Configuration and settings in AWS AppSync

AWS AppSync enables you to:

- Cache data that's requested often but unlikely to change from request to request. This can reduce the load on your resolvers. For more information, see the section called “Configuring server-side caching and API payload compression”.
- Version GraphQL objects to handle and avoid conflict among multiple clients. For more information, see the section called “Versioning, conflict detection, and sync operations for DynamoDB”.
- Use custom domain names to configure a single, memorable domain that works for both your GraphQL and real-time APIs. For more information, see Configuring custom domain names.
- Allow access to your GraphQL APIs through a VPC. For more information, see Using AWS AppSync Private APIs.
- Share your GraphQL APIs through an integration with AWS Resource Access Manager. For more information, see Sharing your AWS AppSync APIs.
- Enable introspection and set query depth and resolver limits per query. For more information, see Configuration limits.
- Use environment variables to adjust your AWS AppSync resolvers' and functions' behavior without updating your code. For more information, see Using environment variables in AWS AppSync.

Additionally, AWS AppSync includes the following standard AWS tools for logging, monitoring, and tracing:

- Logging in AWS CloudTrail
- Monitoring with Amazon CloudWatch
- Tracing with AWS X-Ray

Configuring server-side caching and API payload compression in AWS AppSync

AWS AppSync's server-side data caching capabilities make data available in a high speed, in-memory cache, improving performance and decreasing latency. This reduces the need to directly access data sources. Caching is available for both unit and pipeline resolvers.

AWS AppSync also allows you to compress API responses so that payload content loads and downloads faster. This potentially reduces the strain on your applications while also potentially reducing your data transfer charges. Compression behavior is configurable and can be set at your own discretion.

Refer to this section for help defining the desired behavior of server-side caching and compression in your AWS AppSync API.

Instance types

AWS AppSync hosts Amazon ElastiCache (Redis OSS) instances in the same AWS account and AWS Region as your AWS AppSync API.

The following ElastiCache (Redis OSS) instance types are available:

small

1 vCPU, 1.5 GiB RAM, low to moderate network performance

medium

2 vCPU, 3 GiB RAM, low to moderate network performance

large

2 vCPU, 12.3 GiB RAM, up to 10 Gigabit network performance

xlarge

4 vCPU, 25.05 GiB RAM, up to 10 Gigabit network performance

2xlarge

8 vCPU, 50.47 GiB RAM, up to 10 Gigabit network performance

4xlarge

16 vCPU, 101.38 GiB RAM, up to 10 Gigabit network performance

8xlarge

32 vCPU, 203.26 GiB RAM, 10 Gigabit network performance (not available in all Regions)

12xlarge

48 vCPU, 317.77 GiB RAM, 10 Gigabit network performance

> **Note**
>
> Historically, you specified a specific instance type (such as `t2.medium`). As of July 2020, these legacy instance types continue to be available, but their use is deprecated and discouraged. We recommend that you use the generic instance types described here.

Caching behavior

The following are the behaviors related to caching:

None

No server-side caching.

Full request caching

Full request caching is a mechanism that caches resolver execution results individually. With this setting, AWS AppSync caches the execution of all resolvers invoked during a request, with each resolver cached separately. The data for each resolver is retrieved from its data source and populates the cache until the time to live (TTL) expires. For subsequent API requests, results for each specific resolver are returned from the cache. This means that data sources aren't contacted directly unless the TTL has expired. AWS AppSync uses the contents of the `context.arguments` and `context.identity` maps as caching keys for each resolver.

Per-resolver caching

With this setting, each resolver must be explicitly opted in for it to cache responses. You can specify a TTL and caching keys on the resolver. Caching keys that you can specify are the top-level maps `context.arguments`, `context.source`, and `context.identity`, and/or string fields from these maps. The TTL value is mandatory, but the caching keys are optional. If you don't specify any caching keys, the defaults are the contents of the `context.arguments`, `context.source`, and `context.identity` maps.

For example, you could use the following combinations:

- *context.arguments* and *context.source*
- *context.arguments* and *context.identity.sub*
- *context.arguments.id* or *context.arguments.InputType.id*
- *context.source.id* and *context.identity.sub*
- *context.identity.claims.username*

When you specify only a TTL and no caching keys, the behavior of the resolver is the same as full request caching.

Operation level caching

Operation level caching stores entire GraphQL query operation responses as a whole. When enabled, successful query responses are cached until their TTL expires, with a maximum cacheable response size of 15 MB. For subsequent query requests with the same cache key, responses will be served directly from the cache without executing any resolvers while the TTL has not expired.

The cache key for operation level caching is generated using a combination of the following:

- Certain attributes from the request's JSON payload:
  - The `query` string
  - The `operationName` string
  - The `variables` map
- The `context.identity` map (excluding `context.identity.sourceIp` for IAM and Amazon Cognito requests)
- The `context.request.headers` map (excluding certain reserved headers that are listed in the next section)

The authorization type used by the request will also affect the cache key. For IAM-authorized requests, the cache key will additionally include the list of allowed and denied resources. For Lambda-authorized requests, the cache key will additionally include the list of denied fields.

The cache key will consider all request headers found in `context.request.headers`, except the following reserved headers, which are typically unique to specific requests:

- 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

Cache time to live

This setting defines the amount of time to store cached entries in memory. The maximum TTL is 3,600 seconds (1 hour), after which entries are automatically deleted.

Cache encryption

When you use AWS AppSync's server-side data caching feature, encryption at rest and in transit is always enabled for new caches, and can't be disabled.

To enable encryption on an existing API cache, delete the cache and then recreate it.

To invalidate cache entries, you can make a flush cache API call using either the AWS AppSync console or the AWS Command Line Interface (AWS CLI).

For more information, see the ApiCache data type in the AWS AppSync API Reference.

Cache eviction

When you set up AWS AppSync's server-side caching, you can configure a maximum TTL. This value defines the amount of time that cached entries are stored in memory. In situations where you must remove specific entries from your cache, you can use AWS AppSync's `evictFromApiCache` extensions utility in your resolver's request or response. (For example, when your data in your data sources have changed, and your cache entry is now stale.) To evict an item from the cache, you must know its key. For this reason, if you must evict items dynamically, we recommend using per-resolver caching and explicitly defining a key to use to add entries to your cache.

Evicting a cache entry

To evict an item from the cache, use the `evictFromApiCache` extensions utility. Specify the type name and field name, then provide an object of key-value items to build the key of the entry that you want to evict. In the object, each key represents a valid entry from the `context` object that is used in the cached resolver's `cachingKey` list. Each value is the actual value used to construct the value of the key. You must put the items in the object in the same order as the caching keys in the cached resolver's `cachingKey` list.

For example, see the following schema:

```
type Note {
  id: ID!
  title: String
  content: String!
}

type Query {
  getNote(id: ID!): Note
}

type Mutation {
  updateNote(id: ID!, content: String!): Note
```

```
}
```

In this example, you can enable per-resolver caching, then enable it for the `getNote` query. Then, you can configure the caching key to consist of `[context.arguments.id]`.

When you try to get a `Note`, to build the cache key, AWS AppSync performs a lookup in its server-side cache using the `id` argument of the `getNote` query.

When you update a `Note`, you must evict the entry for the specific note to make sure that the next request fetches it from the backend data source. To do this, you must create a request handler.

The following example shows one way to handle the eviction using this method:

```
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;
```

Alternatively, you can also handle the eviction in the response handler.

When the `updateNote` mutation is processed, AWS AppSync tries to evict the entry. If an entry is successfully cleared, the response contains an `apiCacheEntriesDeleted` value in the `extensions` object that shows how many entries were deleted:

```
"extensions": {  "apiCacheEntriesDeleted": 1}
```

Evicting a cache entry based on identity

You can create caching keys based on multiple values from the `context` object.

For example, take the following schema that uses Amazon Cognito user pools as the default auth mode and is backed by an Amazon DynamoDB data source:

```
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
}
```

The `Note` object types are saved in a DynamoDB table. The table has a composite key that uses the Amazon Cognito user name as the primary key and the `id` (a slug) of the `Note` as the partition key. This is a multi-tenant system that allows multiple users to host and update their private `Note` objects, which are never shared.

Since this is a read-heavy system, the `getNote` query is cached using per-resolver caching, with the caching key composed of `[context.identity.username, context.arguments.id]`. When a `Note` is updated, you can evict the entry for that specific `Note`. You must add the components in the object in the same order that they are specified in your resolver's `cachingKeys` list.

The following example shows this:

```
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;
```

A backend system can also update the `Note` and evict the entry. For example, take this mutation:

```
type Mutation {
  updateNoteFromBackend(id: ID!, content: String!, username: ID!): Note @aws_iam
}
```

You can evict the entry, but add the components of the caching key to the `cachingKeys` object.

In the following example, the eviction occurs in the response of the resolver:

```
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;
}
```

In cases where your backend data has been updated outside of AWS AppSync, you can evict an item from the cache by calling a mutation that uses a `NONE` data source.

Compressing API responses

AWS AppSync allows clients to request compressed payloads. If requested, API responses are compressed and returned in response to requests that indicate that compressed content is preferred. Compressed API responses load faster, content is downloaded faster, and your data transfer charges may be reduced as well.

> **Note**
>
> Compression is available on all new APIs created after June 1st, 2020.
> AWS AppSync compresses objects on a best-effort basis. In rare cases, AWS AppSync may skip compression based on a variety of factors, including current capacity.

AWS AppSync can compress GraphQL query payload sizes between 1,000 to 10,000,000 bytes. To enable compression, a client must send the `Accept-Encoding` header with the value `gzip`. Compression can be verified by checking the `Content-Encoding` header's value in the response (`gzip`).

The query explorer in the AWS AppSync console automatically sets the header value in the request by default. If you execute a query that has a large enough response, compression can be confirmed using your browser developer tools.

Configuring custom domain names for GraphQL and real-time APIs

With AWS AppSync, you can use custom domain names to configure a single, memorable domain that works for both your GraphQL and real-time APIs.

In other words, you can utilize simple and memorable endpoint URLs with domain names of your choice by creating custom domain names that you associate with the AWS AppSync APIs in your account.

When you configure an AWS AppSync API, two endpoints are provisioned:

AWS AppSync GraphQL endpoint:

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync real-time endpoint:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/
graphql
```

With custom domain names, you can interact with both endpoints using a single domain. For example, if you configure `api.example.com` as your custom domain, you can interact with both your GraphQL and real-time endpoints using these URLs:

AWS AppSync custom domain GraphQL endpoint:

```
https://api.example.com/graphql
```

AWS AppSync custom domain real-time endpoint:

```
wss://api.example.com/graphql/realtime
```

> **Note**
>
> AWS AppSync APIs support only TLS 1.2 and TLS 1.3 for custom domain names.

Registering and configuring a domain name

To set up custom domain names for your AWS AppSync APIs, you must have a registered internet domain name. You can register an internet domain using Amazon Route 53 domain registration or a third-party domain registrar of your choice. For more information about Route 53, see What is Amazon Route 53? in the *Amazon Route 53 Developer Guide*.

An API's custom domain name can be the name of a subdomain or the root domain (also known as the "zone apex") of a registered internet domain. After you create a custom domain name in AWS AppSync, you must create or update your DNS provider's resource record to map to your API endpoint. Without this mapping, API requests bound for the custom domain name cannot reach AWS AppSync.

Creating a custom domain name in AWS AppSync

Creating a custom domain name for an AWS AppSync API sets up an Amazon CloudFront distribution. You must set up a DNS record to map the custom domain name to the CloudFront distribution domain name. This mapping is required to route API requests that are bound for the custom domain name AWS AppSync through the mapped CloudFront distribution. You must also provide a certificate for the custom domain name.

To set up the custom domain name or to update its certificate, you must have permission to update CloudFront distributions and describe the AWS Certificate Manager (ACM) certificate that you plan to use. To grant these permissions, attach the following AWS Identity and Access Management (IAM) policy statement to an IAM user, group, or role in your account:

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:us-east-1:111122223333:certificate/
certificate-id"
        }
    ]
}
```

AWS AppSync supports custom domain names by leveraging Server Name Indication (SNI) on the CloudFront distribution. For more information about using custom domain names on a CloudFront distribution, including the required certificate format and the maximum certificate key length, see Using HTTPS with CloudFront in the *Amazon CloudFront Developer Guide*.

To set up a custom domain name as the API's hostname, the API owner must provide a public or imported ACM certificate in the `us-east-1` *AWS Region (US East (N. Virginia))* that covers the custom domain name. For more information about ACM, see What is AWS Certificate Manager? in the *AWS Certificate Manager User Guide*.

Wildcard custom domain names in AWS AppSync

AWS AppSync supports wildcard custom domain names. To configure a wildcard custom domain name, specify a wildcard character (*) as the first subdomain of a custom domain. This represents all possible subdomains of the root domain. For example, the wildcard custom domain name `*.example.com` results in subdomains such as `a.example.com`, `b.example.com`, and `c.example.com`. All these subdomains route to the same domain.

To use a wildcard custom domain name in AWS AppSync, you must provide a certificate issued by ACM containing a wildcard name that can protect several sites in the same domain. For more information, see ACM certificate characteristics in the *AWS Certificate Manager User Guide*.

Versioning, conflict detection, and sync operations for DynamoDB data sources in AWS AppSync

AWS AppSync's advanced data management capabilities for DynamoDB leverages three key features: versioned data sources, conflict detection and resolution, and sync operations. These

tools enable robust, scalable applications that efficiently handle concurrent data modifications and synchronization in distributed environments.

Versioned data sources form the foundation of AWS AppSync's data management system. This feature automatically enhances DynamoDB items with versioning metadata, records changes made by AWS AppSync mutations to a Delta table, and maintains "tombstones" for deleted items. Developers can configure retention periods for deleted items and change logs, optimizing storage while ensuring data consistency. Versioned data sources streamline the implementation of conflict detection and sync operations, providing a solid base for advanced data handling.

Conflict detection and resolution mechanisms safeguard data integrity when concurrent writes occur. AWS AppSync offers three strategies: Optimistic Concurrency, Automerge, and Lambda-based resolution. Optimistic Concurrency rejects conflicting mutations, allowing clients to retry with updated data. Automerge automatically resolves conflicts based on data types, merging lists, performing set unions, and preserving existing scalar values. Lambda-based resolution enables custom logic for complex conflict scenarios. These options give developers flexibility in handling data conflicts, ensuring consistency across distributed systems.

Sync operations enable efficient data retrieval and updates in client applications. This feature allows clients to fetch all results from a DynamoDB table and subsequently retrieve only data altered since their last query. AWS AppSync determines whether to access the Base table or Delta table based on the provided sync token, optimizing performance and reducing data transfer.

Topics

Versioning DynamoDB data sources in AWS AppSync

AWS AppSync currently supports versioning on DynamoDB data sources. Conflict Detection, Conflict Resolution, and Sync operations require a `Versioned` data source. When you enable versioning on a data source, AWS AppSync will automatically:

- Enhance items with object versioning metadata.
- Record changes made to items with AWS AppSync mutations to a *Delta* table.
- Maintain deleted items in the *Base* table with a “tombstone” for a configurable amount of time.

Versioned data source configuration

When you enable versioning on a DynamoDB data source, you specify the following fields:

`BaseTableTTL`

The number of minutes to retain deleted items in the *Base* table with a “tombstone” - a metadata field indicating that the item has been deleted. You can set this value to *0* if you want items to be removed immediately when they are deleted. This field is required.

`DeltaSyncTableName`

The name of the table where changes made to items with AWS AppSync mutations are stored. This field is required.

`DeltaSyncTableTTL`

The number of minutes to retain items in the *Delta* table. This field is required.

Delta sync table logging

AWS AppSync currently supports Delta Sync Logging for mutations using `PutItem`, `UpdateItem`, and `DeleteItem` DynamoDB operations.

When an AWS AppSync mutation changes an item in a versioned data source, a record of that change will be stored in a *Delta* table that is optimized for incremental updates. You can choose to use different *Delta* tables (e.g. one per type, one per domain area) for other versioned data sources or a single *Delta* table for your API. AWS AppSync recommends against using a single *Delta* table for multiple APIs to avoid the collision of primary keys.

The schema required for this table is as follows:

`ds_pk`

A string value that is used as the partition key. It is constructed by concatenating the *Base* data source name and the ISO 8601 format of the date the change occurred (e.g. `Comments:2019-01-01`).

When the `customPartitionKey` flag from the VTL mapping template is set as the column name of the partition key (see Resolver Mapping Template Reference for DynamoDB in the *AWS AppSync Developer Guide*), the format of `ds_pk` changes, and the string is constructed by

appending it the value of the partition key in the new record in the *Base* table. For example, if the record in the *Base* table has a partition key value of `1a` and a sort key value of `2b`, the new value of the string will be: `Comments:2019-01-01:1a`.

`ds_sk`

A string value that is used as the sort key. It is constructed by concatenating the ISO 8601 format of the time the change occurred, the primary key of the item, and the version of the item. The combination of these fields guarantees uniqueness for every entry in the *Delta* table (e.g. for a time of `09:30:00` and an ID of `1a` and version of `2`, this would be `09:30:00:1a:2`).

When the `customPartitionKey` flag from the VTL mapping template is set to the column name of the partition key (see Resolver Mapping Template Reference for DynamoDB in the *AWS AppSync Developer Guide*), the format of `ds_sk` changes, and the string is constructed by replacing the value of the combination key with the value of the sort key in the *Base* table. Using the previous example above, if the record in the *Base* table has a partition key value of `1a` and a sort key value of `2b`, the new value of the string will be: `09:30:00:2b:3`.

`_ttl`

A numeric value that stores the timestamp, in epoch seconds, when an item should be removed from the *Delta* table. This value is determined by adding the `DeltaSyncTableTTL` value configured on the data source to the moment when the change occurred. This field should be configured as the DynamoDB TTL Attribute.

The IAM role configured for use with the *Base* table must also contain permission to operate on the *Delta* table. In this example, the permissions policy for a *Base* table called `Comments` and a *Delta* table called `ChangeLog` is displayed:

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/*"
            ]
        }
    ]
}
```

Versioned data source metadata

AWS AppSync manages metadata fields on `Versioned` data sources on your behalf. Modifying these fields yourself may cause errors in your application or data loss. These fields include:

`_version`

A monotonically increasing counter that is updated any time that a change occurs to an item.

`_lastChangedAt`

A numeric value that stores the timestamp, in epoch milliseconds, when an item was last modified.

`_deleted`

A Boolean “tombstone” value that indicates that an item has been deleted. This can be used by applications to evict deleted items from local data stores.

`_ttl`

A numeric value that stores the timestamp, in epoch seconds, when an item should be removed from the underlying data source.

`ds_pk`

A string value that is used as the partition key for *Delta* tables.

`ds_sk`

A string value that is used as the sort key for *Delta* tables.

`gsi_ds_pk`

A string value attribute that's generated to support a global secondary index as a partition key. It will be included only if both the `customPartitionKey` and `populateIndexFields` flags are enabled in the VTL mapping template (see Resolver Mapping Template Reference for DynamoDB in the *AWS AppSync Developer Guide*). If enabled, the value will be constructed by concatenating the *Base* data source name and the ISO 8601 format of the date at which the change occurred (e.g. if the *Base* table is named *Comments*, this record will be set as `Comments:2019-01-01`).

`gsi_ds_sk`

A string value attribute that's generated to support a global secondary index as a sort key. It will be included only if both the `customPartitionKey` and `populateIndexFields` flags are enabled in the VTL mapping template (see Resolver Mapping Template Reference for DynamoDB in the *AWS AppSync Developer Guide*). If enabled, the value will be constructed by concatenating the ISO 8601 format of the time at which the change occurred, the partition key of the item in the *Base* table, the sort key of the item in the *Base* table, and the version of the item (e.g. for a time of `09:30:00`, a partition key value of `1a`, a sort key value of `2b`, and version of `3`, this would be `09:30:00:1a#2b:3`).

These metadata fields will impact the overall size of items in the underlying data source. AWS AppSync recommends reserving *500 bytes + Max Primary Key Size* of storage for versioned data source metadata when designing your application. To use this metadata in client applications, include the `_version`, `_lastChangedAt`, and `_deleted` fields on your GraphQL types and in the selection set for mutations.

Conflict detection and resolution in AWS AppSync

When concurrent writes happen with AWS AppSync, you can configure Conflict Detection and Conflict Resolution strategies to handle updates appropriately. Conflict Detection determines if the mutation is in conflict with the actual written item in the data source. Conflict Detection is enabled by setting the value in the SyncConfig for the `conflictDetection` field to `VERSION`.

Conflict Resolution is the action that is taken in the event that a conflict is detected. This is determined by setting the Conflict Handler field in the SyncConfig. There are three Conflict Resolution strategies:

- OPTIMISTIC_CONCURRENCY

- AUTOMERGE
- LAMBDA

Versions are automatically incremented by AWS AppSync during write operations and should not be modified by clients or outside of a resolver configured with a version-enabled data source. Doing so will change the consistency behavior of the system and could result in data loss.

Optimistic concurrency

Optimistic Concurrency is a conflict resolution strategy that AWS AppSync provides for versioned data sources. When the conflict resolver is set to Optimistic Concurrency, if an incoming mutation is detected to have a version that differs from the actual version of the object, the conflict handler will simply reject the incoming request. Inside the GraphQL response, the existing item on the server that has the latest version will be provided. The client is then expected to handle this conflict locally and retry the mutation with the updated version of the item.

Automerges

Automerge provides developers an easy way to configure a conflict resolution strategy without writing client-side logic to manually merge conflicts that were unable to be handled by other strategies. Automerge adheres to a strict rule set when merging data to resolve conflicts. The tenets of Automerge revolve around the underlying data type of the GraphQL field. They are as follows:

- Conflict on a scalar field: GraphQL scalar or any field that is not a collection (i.e. List, Set, Map). Reject the incoming value for the scalar field and select the value existing in the server.
- Conflict on a list: GraphQL type and database type are lists. Concatenate the incoming list with the existing list in the server. The list values in the incoming mutation will be appended to the end of the list in the server. Duplicate values will be retained.
- Conflict on a set: GraphQL type is a list and database type is a Set. Apply a set union using incoming the set and the existing set in the server. This adheres to the properties of a Set, meaning no duplicate entries.
- When an incoming mutation adds a new field to the item or is made against a field with the value of `null`, merge that on to the existing item.
- Conflict on a map: When the underlying data type in the database is a Map (i.e. key-value document), apply the above rules as it parses and processes each property of the Map.

Automerge is designed to automatically detect, merge, and retry requests with an updated version, absolving the client from needing to manually merge any conflicting data.

To show an example of how Automerge handles a Conflict on a Scalar type. We will use the following record as our starting point.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 4
}
```

Now an incoming mutation might be attempting to update the item but with an older version since the client has not synchronized with the server yet. That looks like this:

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 55,
  "_version" : 2
}
```

Notice the outdated version of 2 in the incoming request. During this flow, Automerge will merge the data by rejecting the 'jersey' field update to '55' and keep the value at '5' resulting in the following image of the item being saved in the server.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 5 # version is incremented every time automerge performs a merge that is
 stored on the server.
}
```

Given the state of the item shown above at version 5, now suppose an incoming mutation that attempts to mutate the item with the following image:

```
{
  "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
}
```

There are three points of interest in the incoming mutation. The name, a scalar, has been changed but two new fields “interests”, a Set, and “points”, a List, have been added. In this scenario, a conflict will be detected due to the version mismatch. Automerge adheres to its properties and rejects the name change due to it being a scalar and add on the non-conflicting fields. This results in the item that is saved in the server to appear as follows.

```
{
  "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
}
```

With the updated image of the item with version 6, now suppose an incoming mutation (with another version mismatch) tries to transform the item to the following:

```
{
  "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
}
```

Here we observe that the incoming field for “interests” has one duplicate value that exists in the server and two new values. In this case, since the underlying data type is a Set, Automerge will combine the values existing in the server with the ones in the incoming request and strip out any duplicates. Similarly there is a conflict on the “points” field where there is one duplicate value and one new value. But since the underlying data type here is a List, Automerge will simply append all values in the incoming request to the end of the values already existing in the server. The resulting merged image stored on the server would appear as follows:

```
{
  "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
}
```

Now let's assume the item stored in the server appears as follows, at version 8.

```
{
  "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
}
```

But an incoming request tries to update the item with the following image, once again with a version mismatch:

```
{
  "id" : 1,
  "name" : "Nadia",
  "stats": {
      "ppg": "25.7",
      "rpg": "6.9"
  }
  "_version" : 3
}
```

Now in this scenario, we can see that the fields that already exist in the server are missing (interests, points, jersey). In addition, the value for "ppg" within the map "stats" is being edited,

a new value “rpg” is being added, and “apg” is omitted. Automerge preserve the fields that have been omitted (note: if fields are intended to be removed, then the request must be tried again with the matching version), and so they will not be lost. It will also apply the same rules to fields within maps and therefore the change to “ppg” will be rejected whereas “apg” is preserved and “rpg”, a new field”, is added on. The resulting item stored in the server will now appear as:

```
{
  "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
}
```

Lambdas

There are several Lambda resolution strategies to choose from:

- `RESOLVE`: Replaces the existing item with new item supplied in response payload. You can only retry the same operation on a single item at a time. Currently supported for DynamoDB `PutItem` & `UpdateItem`.
- `REJECT`: Rejects the mutation and returns an error with the existing item in the GraphQL response. Currently supported for DynamoDB `PutItem`, `UpdateItem`, & `DeleteItem`.
- `REMOVE`: Removes the existing item. Currently supported for DynamoDB `DeleteItem`.

The Lambda Invocation Request

The AWS AppSync DynamoDB resolver invokes the Lambda function specified in the `LambdaConflictHandlerArn`. It uses the same `service-role-arn` configured on the data source. The payload of the invocation has the following structure:

```
{
```

```
    "newItem": { ... },
    "existingItem": {... },
    "arguments": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

The fields are defined as follows:

`newItem`

The preview item, if the mutation succeeded.

`existingItem`

The item currently resided in DynamoDB table.

`arguments`

The arguments from the GraphQL mutation.

`resolver`

Information about the AWS AppSync resolver.

`identity`

Information about the caller. This field is set to null, if access with API key.

Example payload:

```
{
    "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"
    }
}
```

The Lambda Invocation Response

For `PutItem` and `UpdateItem` conflict resolution

`RESOLVE` the mutation. The response must be in the following format.

```
{
    "action": "RESOLVE",
    "item": { ... }
}
```

The `item` field represents an object that will be used to replace the existing item in the underlying data source. The primary key and sync metadata will be ignored if included in `item`.

`REJECT` the mutation. The response must be in the following format.

```
{
    "action": "REJECT"
}
```

For `DeleteItem` conflict resolution

REMOVE the item. The response must be in the following format.

```
{
    "action": "REMOVE"
}
```

REJECT the mutation. The response must be in the following format.

```
{
    "action": "REJECT"
}
```

The example Lambda function below checks who makes the call and the resolver name. If it is made by `jeffTheAdmin`, REMOVE the object for DeletePost resolver or RESOLVE the conflict with new item for Update/Put resolvers. If not, the mutation is REJECT.

```
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;
}
```

Errors

Below is a list of possible errors that may occur during a Conflict Resolution process:

`ConflictUnhandled`

Conflict detection finds a version mismatch and the conflict handler rejects the mutation.

Example: Conflict resolution with an Optimistic Concurrency conflict handler. Or, Lambda conflict handler returned with `REJECT`.

`ConflictError`

An internal error occurs when trying to resolve a conflict.

Example: Lambda conflict handler returned a malformed response. Or, cannot invoke Lambda conflict handler because the supplied resource `LambdaConflictHandlerArn` is not found.

`MaxConflicts`

Max retry attempts were reached for conflict resolution.

Example: Too many concurrent requests on the same object. Before the conflict is resolved, the object is updated to a new version by another client.

`BadRequest`

Client tries to update metadata fields (`_version`, `_ttl`, `_lastChangedAt`, `_deleted`).

Example: Client tries to update `_version` of an object with an update mutation.

`DeltaSyncWriteError`

Failed to write delta sync record.

Example: Mutation succeeded, but an internal error occurred when trying to write to the delta sync table.

`InternalFailure`

An internal error occurred.

`UnsupportedOperation`

Unsupported operation '*X*'. Datasource Versioning only supports the following operations (TransactGetItems,PutItem,BatchGetItem,Scan,Query,GetItem,DeleteItem,UpdateItem,Sync).

Example: Using certain transaction and batch operations with conflict detection/resolution enabled. These operations are not currently supported.

CloudWatch Logs

If an AWS AppSync API has enabled CloudWatch Logs with the logging settings set to Field-Level Logs `enabled` and log-level for the Field-Level Logs set to `ALL`, then AWS AppSync will emit Conflict Detection and Resolution information to the log group. For information about the format of the log messages, see the documentation for Conflict Detection and Sync Logging.

Using DynamoDB sync operations on versioned data sources in AWS AppSync

Versioned data sources support `Sync` operations that allow you to retrieve all the results from a DynamoDB table and then receive only the data altered since your last query (the delta updates). When AWS AppSync receives a request for a `Sync` operation, it uses the fields specified in the request to determine if the *Base* table or the *Delta* table should be accessed.

- If the `lastSync` field is not specified, a `Scan` on the *Base* table is performed.
- If the `lastSync` field is specified, but the value is before the `current moment - DeltaSyncTTL`, a `Scan` on the *Base* table is performed.
- If the `lastSync` field is specified, and the value is on or after the `current moment - DeltaSyncTTL`, a `Query` on the *Delta* table is performed.

AWS AppSync returns the `startedAt` field to the response mapping template for all `Sync` operations. The `startedAt` field is the moment, in epoch milliseconds, when the `Sync` operation started that you can store locally and use in another request. If a pagination token was included in the request, this value will be the same as the one returned by the request for the first page of results.

For information about the format for Sync mapping templates, see the mapping template reference.

Using CloudWatch to monitor and log GraphQL API data

You can log and debug your GraphQL API using CloudWatch metrics and CloudWatch logs. These tools enable developers to monitor performance, troubleshoot issues, and optimize their GraphQL operations effectively.

CloudWatch metrics is a tool that provides a wide range of metrics to monitor API performance and usage. These metrics fall into two main categories:

1. **General API Metrics**: These include `4XXError` and `5XXError` for tracking client and server errors, `Latency` for measuring response times, `Requests` for monitoring total API calls, and `TokensConsumed` for tracking resource usage.
2. **Real-time Subscription Metrics**: These metrics focus on WebSocket connections and subscription activities. They include metrics for connection requests, successful connections, subscription registrations, message publishing, and active connections and subscriptions.

The guide also introduces Enhanced Metrics, which offer more granular data on resolver performance, data source interactions, and individual GraphQL operations. These metrics provide deeper insights but come with additional costs.

CloudWatch Logs is a tool that enables logging capabilities for your GraphQL APIs. Logs can be set at two levels of the API:

1. **Request-level Logs**: These capture overall request information, including HTTP headers, GraphQL queries, operation summaries, and subscription registrations.
2. **Field-level Logs**: These provide detailed information about individual field resolutions, including request and response mappings, and tracing information for each field.

You can configure logging, interpret log entries, and use log data for troubleshooting and optimization. AWS AppSync provides various log types that reveal your query's execution, parsing, validation, and field resolution data.

Setup and configuration

To turn on automatic logging on a GraphQL API, use the AWS AppSync console.

1. Sign in to the AWS Management Console and open the AppSync console.
2. On the **APIs** page, choose the name of a GraphQL API.
3. On your API's homepage, in the navigation pane, choose **Settings**.
4. Under **Logging**, do the following:

    a. Turn on **Enable Logs**.

    b. For detailed request-level logging, select the check box under **Include verbose content**. (optional)

    c. Under **Field resolver log level**, choose your preferred field-level logging level (**None**, **Error**, **Info**, **Debug**, or **All**). (optional)

    d. Under **Create or use an existing role**, choose **New role** to create a new AWS Identity and Access Management (IAM) that allows AWS AppSync to write logs to CloudWatch. Or, choose **Existing role** to select the Amazon Resource Name (ARN) of an existing IAM role in your AWS account.
5. Choose **Save**.

Manual IAM role configuration

If you choose to use an existing IAM role, the role must grant AWS AppSync the required permissions to write logs to CloudWatch. To configure this manually, you must provide a service role ARN so that AWS AppSync can assume the role when writing the logs.

In the IAM console, create a new policy with the name `AWSAppSyncPushToCloudWatchLogsPolicy` that has the following definition:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
```

```
            "Resource": "*"
        }
    ]
}
```

Next, create a new role with the name **AWSAppSyncPushToCloudWatchLogsRole**, and attach the newly created policy to the role. Edit the trust relationship for this role to the following:

JSON

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

Copy the role ARN and use it when setting up logging for an AWS AppSync GraphQL API.

CloudWatch metrics

You can use CloudWatch metrics to monitor and provide alerts about specific events that can result in HTTP status codes or from latency. The following metrics are emitted:

Metrics list

`4XXError`

Errors resulting from requests that are not valid due to an incorrect client configuration. Typically, these errors happen anywhere outside of GraphQL processing. For example, these errors can occur when the request includes an incorrect JSON payload or an incorrect query, when the service is throttled, or when the authorization settings are misconfigured.

Unit: *Count*. Use the Sum statistic to get the total occurrences of these errors.

`5XXError`

Errors encountered during the running of a GraphQL query. For example, this can occur when invoking a query for an empty or incorrect schema. It can also occur when the Amazon Cognito user pool ID or AWS Region is not valid. Alternatively, this could also happen if AWS AppSync encounters an issue during processing of a request.

Unit: *Count*. Use the Sum statistic to get the total occurrences of these errors.

`Latency`

The time between when AWS AppSync receives a request from a client and when it returns a response to the client. This doesn't include the network latency encountered for a response to reach the end devices.

Unit: *Millisecond*. Use the Average statistic to evaluate expected latencies.

`Requests`

The number of requests (queries + mutations) that all APIs in your account have processed, by Region.

Unit: *Count*. The number of all requests processed in a particular Region.

`TokensConsumed`

Tokens are allocated to `Requests` based on the amount of resources (processing time and memory used) that a `Request` consumes. Usually, each `Request` consumes one token. However, a `Request` that consumes large amounts of resources is allocated additional tokens as needed.

Unit: *Count*. The number of tokens allocated to requests processed in a particular Region.

`NetworkBandwidthOutAllowanceExceeded`

> **Note**
>
> In the AWS AppSync console, on the cache settings page, the **Cache Health Metrics** option allows you to enable this cache-related health metric.

The network packets dropped because the throughput exceeded the aggregated bandwidth limit. This is useful for diagnosing bottlenecks in a cache

configuration. Data is recorded for a particular API by specifying the `API_Id` in the `appsyncCacheNetworkBandwidthOutAllowanceExceeded` metric.

Unit: *Count*. The number of packets dropped after exceeding the bandwidth limit for an API specified by ID.

`EngineCPUUtilization`

> **Note**
>
> In the AWS AppSync console, on the cache settings page, the **Cache Health Metrics** option allows you to enable this cache-related health metric.

The CPU utilization (percentage) allocated to the Redis OSS process. This is useful for diagnosing bottlenecks in a cache configuration. Data is recorded for a particular API by specifying the `API_Id` in the `appsyncCacheEngineCPUUtilization` metric.

Unit: *Percent*. The CPU percentage currently in use by the Redis OSS process for an API specified by ID.

Real-time subscriptions

All metrics are emitted in one dimension: **GraphQLAPIId**. This means that all metrics are coupled with GraphQL API IDs. The following metrics are related to GraphQL subscriptions over pure WebSockets:

Metrics list

`ConnectRequests`

The number of WebSocket connection requests made to AWS AppSync, including both successful and unsuccessful attempts.

Unit: *Count*. Use the Sum statistic to get the total number of connection requests.

`ConnectSuccess`

The number of successful WebSocket connections to AWS AppSync. It is possible to have connections without subscriptions.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the successful connections.

`ConnectClientError`

The number of WebSocket connections that were rejected by AWS AppSync because of client-side errors. This could imply that the service is throttled or that the authorization settings are misconfigured.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the client-side connection errors.

`ConnectServerError`

The number of errors that originated from AWS AppSync while processing connections. This usually happens when an unexpected server-side issue occurs.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the server-side connection errors.

`DisconnectSuccess`

The number of successful WebSocket disconnections from AWS AppSync.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the successful disconnections.

`DisconnectClientError`

The number of client errors that originated from AWS AppSync while disconnecting WebSocket connections.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the disconnection errors.

`DisconnectServerError`

The number of server errors that originated from AWS AppSync while disconnecting WebSocket connections.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the disconnection errors.

`SubscribeSuccess`

The number of subscriptions that were successfully registered to AWS AppSync through WebSocket. It's possible to have connections without subscriptions, but it's not possible to have subscriptions without connections.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the successful subscriptions.

`SubscribeClientError`

The number of subscriptions that were rejected by AWS AppSync because of client-side errors. This can occur when a JSON payload is incorrect, the service is throttled, or the authorization settings are misconfigured.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the client-side subscription errors.

`SubscribeServerError`

The number of errors that originated from AWS AppSync while processing subscriptions. This usually happens when an unexpected server-side issue occurs.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the server-side subscription errors.

`UnsubscribeSuccess`

The number of unsubscribe requests that were successfully processed.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the successful unsubscribe requests.

`UnsubscribeClientError`

The number of unsubscribe requests that were rejected by AWS AppSync because of client-side errors.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the client-side unsubscribe request errors.

`UnsubscribeServerError`

The number of errors that originated from AWS AppSync while processing unsubscribe requests. This usually happens when an unexpected server-side issue occurs.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the server-side unsubscribe request errors.

`PublishDataMessageSuccess`

The number of subscription event messages that were successfully published.

Unit: *Count*. Use the Sum statistic to get the total of the subscription event messages were successfully published.

`PublishDataMessageClientError`

The number of subscription event messages that failed to publish because of client-side errors.

`Unit`: *Count*. Use the Sum statistic to get the total occurrences of the client-side publishing subscription events errors.

`PublishDataMessageServerError`

The number of errors that originated from AWS AppSync while publishing subscription event messages. This usually happens when an unexpected server-side issue occurs.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the server-side publishing subscription events errors.

`PublishDataMessageSize`

The size of subscription event messages published.

Unit: *Bytes*.

`ActiveConnections`

The number of concurrent WebSocket connections from clients to AWS AppSync in 1 minute.

Unit: *Count*. Use the Sum statistic to get the total opened connections.

`ActiveSubscriptions`

The number of concurrent subscriptions from clients in 1 minute.

Unit: *Count*. Use the Sum statistic to get the total active subscriptions.

`ConnectionDuration`

The amount of time that the connection stays open.

Unit: *Milliseconds*. Use the Average statistic to evaluate connection duration.

`OutboundMessages`

The number of metered messages successfully published. One metered message equals 5 kB of delivered data.

Unit: *Count*. Use the Sum statistic to get the total number of successfully published metered messages.

`InboundMessageSuccess`

The number of inbound messages successfully processed. Each subscription type invoked by a mutation generates one inbound message.

Unit: *Count*. Use the Sum statistic to get the total number of successfully processed inbound messages.

`InboundMessageError`

The number of inbound messages that failed processing due to invalid API requests, such as exceeding the 240 kB subscription payload size limit.

Unit: *Count*. Use the Sum statistic to get the total number of inbound messages with API-related processing failures.

`InboundMessageFailure`

The number of inbound messages that failed processing due to errors from AWS AppSync.

Unit: *Count*. Use the Sum statistic to get the total number of inbound messages with AWS AppSync-related processing failures.

`InboundMessageDelayed`

The number of delayed inbound messages. Inbound messages can be delayed when either the inbound message rate quota or outbound message rate quota is breached.

Unit: *Count*. Use the Sum statistic to get the total number of inbound messages that were delayed.

`InboundMessageDropped`

The number of dropped inbound messages. Inbound messages can be dropped when either the inbound message rate quota or outbound message rate quota is breached.

Unit: *Count*. Use the Sum statistic to get the total number of inbound messages that were dropped.

`InvalidationSuccess`

The number of subscriptions successfully invalidated (unsubscribed) by a mutation with `$extensions.invalidateSubscriptions()`.

Unit: *Count*. Use the Sum statistic to retrieve the total number of subscriptions that were successfully unsubscribed.

`InvalidationRequestSuccess`

The number of invalidation requests successfully processed.

Unit: *Count*. Use the Sum statistic to get the total number of successfully processed invalidation requests.

`InvalidationRequestError`

The number of invalidation requests that failed processing due to invalid API requests.

Unit: *Count*. Use the Sum statistic to get the total number of invalidation requests with API-related processing failures.

`InvalidationRequestFailure`

The number of invalidation requests that failed processing due to errors from AWS AppSync.

Unit: *Count*. Use the Sum statistic to get the total number of invalidation requests with AWS AppSync-related processing failures.

`InvalidationRequestDropped`

The number of invalidation requests dropped when the invalidation request quota was exceeded.

Unit: *Count*. Use the Sum statistic to get the total number of dropped invalidation requests.

Comparing inbound and outbound messages

When a mutation is executed, subscription fields with the *@aws_subscribe* directive for that mutation are invoked. Each subscription invocation generates one inbound message. For example, if two subscription fields specify the same mutation in *@aws_subscribe*, then two inbound messages are generated when that mutation is called.

One outbound message equals 5 kB of data delivered to WebSocket clients. For example, sending 15 kB of data to 10 clients results in 30 outbound messages (15 kB * 10 clients / 5 kB per message = 30 messages).

You can request quota increases for either inbound or outbound messages. For more information, see AWS AppSync endpoints and quotas in the *AWS General Reference* guide and the instructions for Requesting a quota increase in the *Service Quotas User Guide*.

Enhanced metrics

Enhanced metrics emit granular data on API usage and performance such as AWS AppSync request and error counts, latency, and cache hits/misses. All enhanced metric data is sent to your CloudWatch account, and you can configure the types of data that will be sent.

> **Note**
>
> Additional charges are applied when using enhanced metrics. For more information, see **detailed monitoring** pricing tiers in Amazon CloudWatch pricing.

These metrics can be found on various settings pages in the AWS AppSync console. On the API settings page, the **Enhanced Metrics** section allows you to enable or disable the following items:

Resolver metrics behavior: These options control how additional metrics for resolvers are collected. You can choose to enable full request resolver metrics (metrics enabled for all resolvers in requests) or per-resolver metrics (metrics only enabled for resolvers where the configuration is set to enabled). The following options are available:

Resolver metrics behavior list

`GraphQL errors per resolver (GraphQLError)`

The number of GraphQL errors that occured per resolver.

Metric dimension: `API_Id`, `Resolver`

Unit: *Count*.

`Requests per resolver (Request)`

The number of invocations that occurred during a request. This is recorded on a per-resolver basis.

Metric dimension: `API_Id`, `Resolver`

Unit: *Count*.

`Latency per resolver (Latency)`

The time to complete a resolver invocation. Latency is measured in milliseconds and is recorded on a per-resolver basis.

Metric dimension: `API_Id`, `Resolver`

Unit: *Millisecond*.

`Cache hits per resolver (CacheHit)`

The number of cache hits during a request. This will only be emitted if a cache is used. Cache hits are recorded on a per-resolver basis.

Metric dimension: `API_Id`, `Resolver`

Unit: *Count*.

`Cache misses per resolver (CacheMiss)`

The number of cache misses during a request. This will only be emitted if a cache is used. Cache misses are recorded on a per-resolver basis.

Metric dimension: `API_Id`, `Resolver`

Unit: *Count*.

Data source metrics behavior: These options control how additional metrics for data sources are collected. You can choose to enable full request data source metrics (metrics enabled for all data sources in requests) or per-data source metrics (metrics only enabled for data sources where the configuration is set to enabled). The following options are available:

Data source metrics behavior list

`Requests per data source (Request)`

The number of invocations that occured during a request. Requests are recorded on a per-data source basis. If full requests are enabled, each data source will have its own entry in CloudWatch.

Metric dimension: `API_Id`, `Datasource`

Unit: *Count*.

`Latency per data source (Latency)`

The time to complete a data source invocation. Latency is recorded on a per-data source basis.

Metric dimension: `API_Id`, `Datasource`

Unit: *Millisecond*.

`Errors per data source (GraphQLError)`

The number of errors that occurred during a data source invocation.

Metric dimension: `API_Id`, `Datasource`

Unit: *Count*.

Operation metrics: Enables GraphQL operation-level metrics.

Operation metrics behavior list

`Requests per operation (Request)`

The number of times a specified GraphQL operation was called.

Metric dimension: `API_Id`, `Operation`

Unit: *Count*.

`GraphQL errors per operation (GraphQLError)`

The number of GraphQL errors that occurred during a specified GraphQL operation.

Metric dimension: `API_Id`, `Operation`

Unit: *Count*.

CloudWatch logs

You can configure two types of logging on any new or existing GraphQL API: request-level and field-level.

Request-level logs

When request-level logging (**Include verbose content**) is configured, the following information is logged:

- The number of tokens consumed

- The request and response HTTP headers
- The GraphQL query that is running in the request
- The overall operation summary
- New and existing GraphQL subscriptions that are registered

Field-level logs

When field-level logging is configured, the following information is logged:

- Generated request mapping with source and arguments for each field
- The transformed response mapping for each field, which includes the data as a result of resolving that field
- Tracing information for each field

If you turn on logging, AWS AppSync manages the CloudWatch Logs. The process includes creating log groups and log streams, and reporting to the log streams with these logs.

When you turn on logging on a GraphQL API and make requests, AWS AppSync creates a log group and log streams under the log group. The log group is named following the `/aws/appsync/apis/{graphql_api_id}` format. Within each log group, the logs are further divided into log streams. These are ordered by **Last Event Time** as logged data is reported.

Every log event is tagged with the **x-amzn-RequestId** of that request. This helps you filter log events in CloudWatch to get all logged information about that request. You can get the RequestId from the response headers of every GraphQL AWS AppSync request.

The field-Level logging is configured with the following log levels:

- **None** - **No field-level logs are captured.**
- **Error - Logs the following information only for the fields that are in the error category:**
    - The error section in the server response
    - Field-level errors
    - The generated request/response functions that got resolved for error fields
- **Info - Logs the following information only for the fields that are in the info and error categories:**
    - Info-level messages

    - The user messages sent through `$util.log.info` and `console.log`
    - Field-level tracing and mapping logs are not shown.
    - If field-level logging is set to `INFO` or higher with verbose-content included, AWS AppSync adds the transformed mapping template logging messages. This will contain any information added to the transformed mapping template, or the output of the resolver or function executed JavaScript code, and should not be used if you are planning to send sensitive information, such as passwords or authorization headers, to downstream data sources and do not want that information in your logs.
- **Debug - Logs the following information only for the fields that are in the debug, info, and error categories:**
    - Debug-level messages
    - The user messages sent through `$util.log.info`, `$util.log.debug`, `console.log`, and `console.debug`
    - Field-level tracing and mapping logs are not shown.
- **All - Logs the following information for all fields in the query:**
    - Field-level tracing information
    - The generated request/response functions that were resolved for each field

Benefits of monitoring

You can use logging and metrics to identify, troubleshoot, and optimize your GraphQL queries. For example, these will help you debug latency issues using the tracing information that is logged for each field in the query. To demonstrate this, suppose you are using one or more resolvers nested in a GraphQL query. A sample field operation in CloudWatch Logs might look similar to the following:

```
{
    "path": [
        "singlePost",
        "authors",
        0,
        "name"
    ],
    "parentType": "Post",
    "returnType": "String!",
    "fieldName": "name",
    "startOffset": 416563350,
```

```
    "duration": 11247
}
```

This might correspond to a GraphQL schema, similar to the following:

```
type Post {
  id: ID!
  name: String!
  authors: [Author]
}

type Author {
  id: ID!
  name: String!
}

type Query {
  singlePost(id:ID!): Post
}
```

In the preceding log results, **path** shows a single item in your data returned from running a query named `singlePost()`. In this example, it's representing the **name** field at the first index (0). The **startOffset** gives an offset from the start of the GraphQL query operation. The **duration** is the total time to resolve the field. These values can be useful to troubleshoot why data from a particular data source might be running slower than expected, or if a specific field is slowing down the entire query. For example, you might choose to increase provisioned throughput for an Amazon DynamoDB table, or remove a specific field from a query that is causing the overall operation to perform poorly.

As of May 8, 2019, AWS AppSync generates log events as fully structured JSON. This can help you use log analytics services such as CloudWatch Logs Insights and Amazon OpenSearch Service to understand the performance of your GraphQL requests and usage characteristics of your schema fields. For example, you can easily identify resolvers with large latencies that may be the root cause of a performance issue. You can also identify the most and least frequently used fields in your schema and assess the impact of deprecating GraphQL fields.

Conflict detection and sync logging

If an AWS AppSync API has logging to CloudWatch Logs configured with the **Field resolver log level** set to **All**, then AWS AppSync emits conflict detection and resolution information to the log

group. This provides granular insight into how the AWS AppSync API responded to a conflict. To help you interpret the response, the following information is provided in the logs:

Metrics list

`conflictType`

Details whether a conflict occurred due to a version mismatch or the customer-supplied condition.

`conflictHandlerConfigured`

States the conflict handler configured on the resolver at the time of the request.

`message`

Provides information on how the conflict was detected and resolved.

`syncAttempt`

The number of tries the server attempted in order to synchronize the data before ultimately rejecting the request.

`data`

If the conflict handler configured is `Automerge`, this field is populated to show what decision `Automerge` took for each field. Actions provided can be:

- **REJECTED** - When `Automerge` rejects the incoming field value in favor of the value in the server.
- **ADDED** - When `Automerge` adds on the incoming field due to no pre-existing value in the server.
- **APPENDED** - When `Automerge` appends the incoming values to the values for the List that exists in the server.
- **MERGED** - When `Automerge` merges the incoming values to the values for the Set that exists in the server.

Using token counts to optimize your requests

Requests that consume less than or equal to 1,500 KB-seconds of memory and vCPU time are allocated one token. Requests with resource consumption greater than 1,500 KB-seconds receive additional tokens. For example, if a request consumes 3,350 KB-seconds, AWS AppSync allocates

three tokens (rounded up to the next integer value) to the request. By default, AWS AppSync allocates a maximum of 5,000 or 10,000 request tokens per second to the APIs in your account, depending upon the AWS Region in which it's deployed. If your APIs each use an average of two tokens per second, you'll be limited to 2,500 or 5,000 requests per second, respectively. If you need more tokens per second than the allotted amount, you can submit a request to increase the default quota for the rate of request tokens. For more information, see AWS AppSync endpoints and quotas in the *AWS General Reference guide* and Requesting a quota increase in the *Service Quotas User Guide*.

A high per-request token count could indicate that there's an opportunity to optimize your requests and improve the performance of your API. Factors that can increase your per-request token count include:

- The size and complexity of your GraphQL schema.
- The complexity of request and response mapping templates.
- The number of resolver invocations per request.
- The amount of data returned from resolvers.
- The latency of downstream data sources.
- Schema and query designs that require successive data source calls (as opposed to parallel or batched calls).
- Logging configuration, particularly field-level and verbose log content.

> **Note**
>
> In addition to AWS AppSync metrics and logs, clients can access the number of tokens consumed in a request via the response header `x-amzn-appsync-TokensConsumed`.

Log size limits

By default, if logging has been enabled, AWS AppSync will send up to 1 MB of logs per request. Logs exceeding this size will be truncated. To reduce log sizes, choose the `ERROR` logging level for field-level logs and disable `VERBOSE` logging, or disable field-level logs entirely if not needed. As an alternative to the `ALL` log level, you can use Enhanced Metrics to obtain metrics on specific resolvers, data sources, or GraphQL operations, or utilize the logging utilities provided by AppSync to log only the necessary information.

Log type reference

RequestSummary

- **requestId:** Unique identifier for the request.
- **graphQLAPIId:** ID of the GraphQL API making the request.
- **statusCode:** HTTP status code response.
- **latency:** End-to-end latency of the request, in nanoseconds, as an integer.

```
{
     "logType": "RequestSummary",
     "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
     "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4",
     "statusCode": 200,
     "latency": 242000000
}
```

ExecutionSummary

- **requestId:** Unique identifier for the request.
- **graphQLAPIId:** ID of the GraphQL API making the request.
- **startTime:** The start timestamp of GraphQL processing for the request, in RFC 3339 format.
- **endTime:** The end timestamp of GraphQL processing for the request, in RFC 3339 format.
- **duration:** The total elapsed GraphQL processing time, in nanoseconds, as an integer.
- **version:** The schema version of the ExecutionSummary.
- **parsing:**
    - **startOffset:** The start offset for parsing, in nanoseconds, relative to the invocation, as an integer.
    - **duration:** The time spent parsing, in nanoseconds, as an integer.
- **validation:**
    - **startOffset:** The start offset for validation, in nanoseconds, relative to the invocation, as an integer.
    - **duration:** The time spent performing validation, in nanoseconds, as an integer.

```
{
    "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"
}
```

Tracing

- **requestId:** Unique identifier for the request.
- **graphQLAPIId:** ID of the GraphQL API making the request.
- **startOffset:** The start offset for field resolution, in nanoseconds, relative to the invocation, as an integer.
- **duration:** The time spent resolving the field, in nanoseconds, as an integer.
- **fieldName:** The name of the field being resolved.
- **parentType:** The parent type of the field being resolved.
- **returnType:** The return type of the field being resolved.
- **path:** A list of path segments, starting at the root of the response and ending with the field being resolved.
- **resolverArn:** The ARN of the resolver used for field resolution. Might not be present on nested fields.

```
{
    "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"
}
```

Analyzing your logs with CloudWatch Logs Insights

The following are examples of queries you can run to get actionable insights into the performance and health of your GraphQL operations. These examples are available as sample queries in the CloudWatch Logs Insights console. In the CloudWatch console, choose **Logs Insights**, select the AWS AppSync log group for your GraphQL API, and then choose **AWS AppSync queries** under **Sample queries**.

The following query returns the top 10 GraphQL requests with maximum tokens consumed:

```
filter @message like "Tokens Consumed"
| parse @message "* Tokens Consumed: *" as requestId, tokens
| sort tokens desc
| display requestId, tokens
| limit 10
```

The following query returns the top 10 resolvers with maximum latency:

```
  fields resolverArn, duration
| filter logType = "Tracing"
| limit 10
| sort duration desc
```

The following query returns the most frequently invoked resolvers:

```
  fields ispresent(resolverArn) as isRes
| stats count() as invocationCount by resolverArn
| filter isRes and logType = "Tracing"
| limit 10
```

```
| sort invocationCount desc
```

The following query returns resolvers with the most errors in mapping templates:

```
  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
```

The following query returns resolver latency statistics:

```
  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
```

The following query returns field latency statistics:

```
  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
```

The results of CloudWatch Logs Insights queries can be exported to CloudWatch dashboards.

Analyze your logs with OpenSearch Service

You can search, analyze, and visualize your AWS AppSync logs with Amazon OpenSearch Service to identify performance bottlenecks and root causes of operational issues. You can identify resolvers with the maximum latency and errors. In addition, you can use OpenSearch Dashboards to create dashboards with powerful visualizations. OpenSearch Dashboards is an open source data visualization and exploration tool available in OpenSearch Service. Using OpenSearch Dashboards, you can continuously monitor the performance and health of your GraphQL operations. For example, you can create dashboards to visualize the P90 latency of your GraphQL requests and drill down into the P90 latencies of each resolver.

When using OpenSearch Service, use **"cwl*"** as the filter pattern to search OpenSearch indexes. OpenSearch Service indexes the logs streamed from CloudWatch Logs with a prefix of **"cwl-"**. To differentiate AWS AppSync API logs from other CloudWatch logs sent to OpenSearch Service, we recommend adding an additional filter expression of `graphQLAPIID.keyword=`*`YourGraphQLAPIID`* to your search.

Log format migration

Log events that AWS AppSync generates on or after May 8, 2019 are formatted as fully structured JSON. To analyze GraphQL requests prior to May 8, 2019, you can migrate older logs to fully structured JSON using a script available in the GitHub Sample. If you need to use the log format prior to May 8, 2019, create a support ticket with the following settings: set **Type** to **Account Management** and then set **Category** to **General Account Question**.

You can also use metric filters in CloudWatch to turn log data into numerical CloudWatch metrics, so that you can graph or set an alarm on them.

Using AWS X-Ray to trace requests in AWS AppSync

You can use AWS X-Ray to trace requests as they are executed in AWS AppSync. You can use X-Ray with AWS AppSync in all AWS Regions where X-Ray is available. X-Ray gives you a detailed overview of an entire GraphQL request. This enables you to analyze latencies in your APIs and their underlying resolvers and data sources. You can use an X-Ray service map to view the latency of a request, including any AWS services that are integrated with X-Ray. You can also configure sampling rules to tell X-Ray which requests to record, and at what sampling rates, according to criteria that you specify.

For more information about sampling in X-Ray, see Configuring Sampling Rules in the AWS X-Ray Console.

Setup and Configuration

You can enable X-Ray tracing for a GraphQL API through the AWS AppSync console.

1. Sign in to the AWS AppSync console.
2. Choose **Settings** from the navigation panel.
3. Under **X-Ray**, turn on **Enable X-Ray**.
4. Choose **Save**. X-Ray tracing is now enabled for your API.

If you're using the AWS CLI or AWS CloudFormation, you can also enable X-Ray tracing when you create a new AWS AppSync API, or update an existing AWS AppSync API, by setting the `xrayEnabled` property to `true`.

When X-Ray tracing is enabled for an AWS AppSync API, an AWS Identity and Access Management service-linked role is automatically created in your account with the appropriate permissions. This allows AWS AppSync to send traces to X-Ray in a secure way.

Tracing Your API with X-Ray

Sampling

By using sampling rules, you can control the amount of data that you record in AWS AppSync, and can modify sampling behavior on the fly without modifying or redeploying your code. For example, this rule samples requests to the GraphQL API with the API ID `3n572shhcpfokwhdnq1ogu59v6`.

- **Rule name** — `test-sample`
- **Priority** — `10`
- **Reservoir size** — `10`
- **Fixed rate** — `10`
- **Service name** — *
- **Service type** — `AWS::AppSync::GraphQLAPI`
- **HTTP method** — *
- **Resource ARN** — `arn:aws:appsync:us-west-2:123456789012:apis/3n572shhcpfokwhdnq1ogu59v6`
- **Host** — *

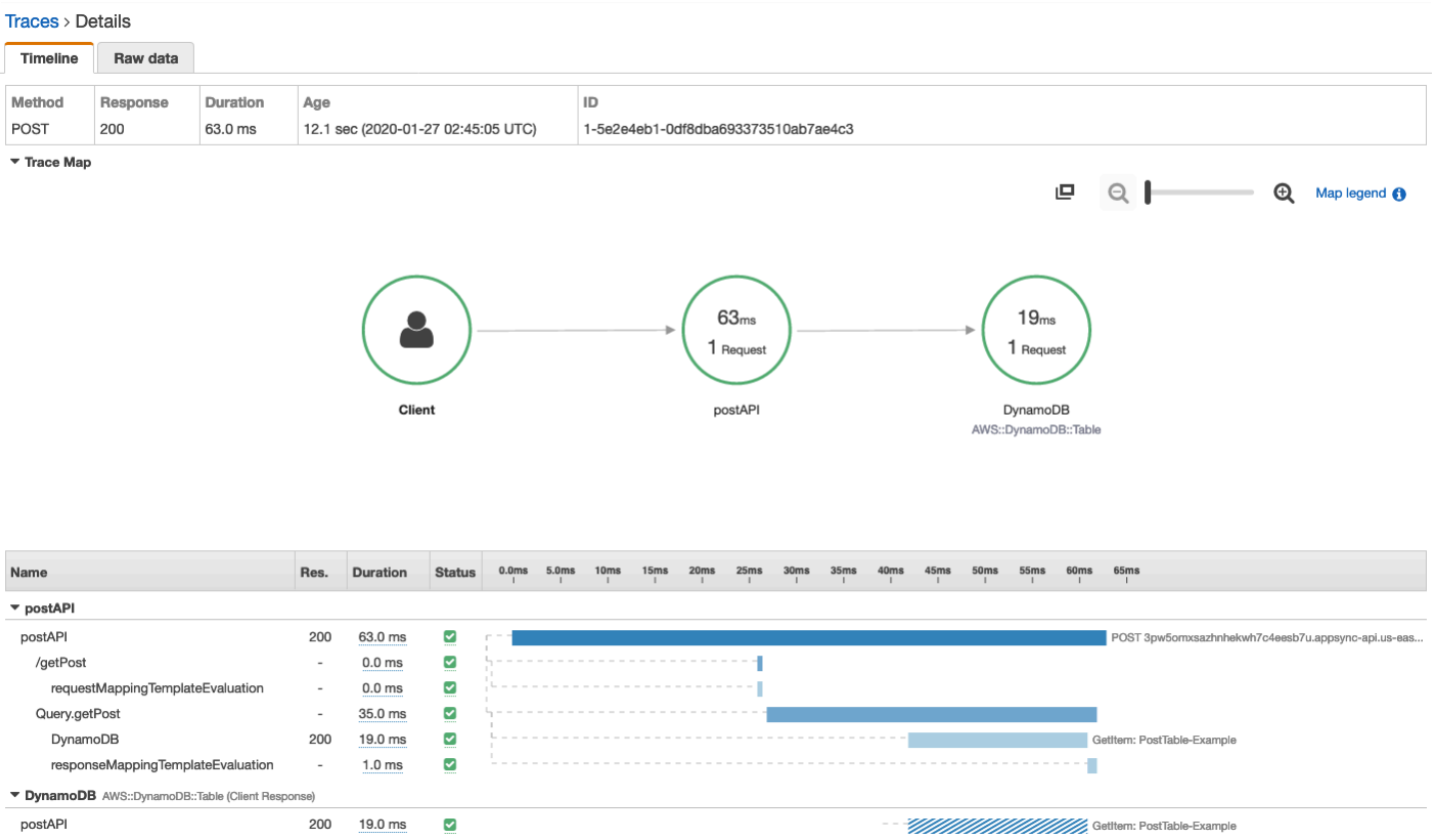
Understanding Traces

When you enable X-Ray tracing for your GraphQL API, you can use the X-Ray trace detail page to examine detailed latency information about requests made to your API. The following example shows the trace view along with the service map for this specific request. The request was made to an API called `postAPI` with a Post type, whose data is contained in an Amazon DynamoDB table called `PostTable-Example`.

The following trace image corresponds to the following GraphQL query:

```
query getPost {
    getPost(id: "1") {
      id
      title
    }
}
```

The resolver for the `getPost` query uses the underlying DynamoDB data source. The following trace view shows the call to DynamoDB, as well as the latencies of various parts of the query's execution:



- In the preceding image, `/getPost` represents the complete path to the element that is being resolved. In this case, because `getPost` is a field on the root `Query` type, it appears directly after the root of the path.
- `requestMappingTemplateEvaluation` represents the time spent by AWS AppSync evaluating the request mapping template for this element in the query.
- `Query.getPost` represents a type and field (in `Type.field` format). It can contain multiple subsegments, depending on the structure of the API and the request being traced.

- `DynamoDB` represents the data source that is attached to this resolver. It contains the latency for the network call to DynamoDB to resolve the field.
- `responseMappingTemplateEvaluation` represents the time spent by AWS AppSync evaluating the response mapping template for this element in the query.

When you view traces in X-Ray, you can get additional contextual and metadata information about the subsegments in the AWS AppSync segment by choosing the subsegments and exploring the detailed view.

For certain deeply nested or complex queries, note that the segment delivered to X-Ray by AWS AppSync can be larger than the maximum size allowed for segment documents, as defined in AWS X-Ray Segment Documents. X-Ray doesn't display segments that exceed the limit.

Logging AWS AppSync API calls using AWS CloudTrail

AWS AppSync is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or AWS service in AWS AppSync. CloudTrail captures all API calls for AWS AppSync as events. The calls captured include calls from the AWS AppSync console and from code calls to the AWS AppSync APIs. You can use the information collected by CloudTrail to determine the request that was made to AWS AppSync, the IP address of the requester, who made the request, when the request was made, and additional details.

You can create a *trail* to enable continuous delivery of CloudTrail events to an Amazon Simple Storage Service (Amazon S3) bucket, including events for AWS AppSync. If you don't configure a trail, you can still view the most recent events in the CloudTrail console.

For more information about CloudTrail, see the AWS CloudTrail User Guide.

AWS AppSync information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. In the CloudTrail console in **Event history**, you can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History in the *AWS CloudTrail User Guide*.

For an ongoing record of events in your AWS account, including events for AWS AppSync, create a trail. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and

act upon the event data collected in CloudTrail logs. For more information, see the following in the *AWS CloudTrail User Guide*:

- Creating a Trail For Your AWS Account
- AWS Service Integrations With CloudTrail Logs
- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions
- Receiving CloudTrail Log Files from Multiple Accounts

CloudTrail logs all AWS AppSync API operations. For example, calls to the `CreateGraphqlApi`, `CreateDataSource`, and `ListResolvers` APIs generate entries in the CloudTrail log files. These and other operations are documented in the AWS AppSync API Reference.

Every event or log entry contains information about who generated the request. The identity information helps you determine:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see CloudTrail userIdentity Element in the *AWS CloudTrail User Guide*.

AWS AppSync data events in CloudTrail

Data events provide information about the resource operations performed on or in a resource (for example, reading or writing to an Amazon S3 object). These are also known as data plane operations. Data events are often high-volume activities. By default, CloudTrail doesn't log data events. The CloudTrail **Event history** doesn't record data events.

Additional charges apply for data events. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

You can log data events for the `AWS::AppSync::GraphQLApi` resource type by using the CloudTrail console, AWS CLI, or CloudTrail API operations (these include query, mutation, and subscription operations, connect operations to your real-time WebSocket endpoint, but not messages sent over your real-time WebSocket endpoint.) For more information about how to log

data events, see Logging data events with the AWS Management Console and Logging data events with the AWS Command Line Interface in the *AWS CloudTrail User Guide*.

The following table lists the AWS AppSync resource type for which you can log data events. The **Data event type (console)** column shows the value to choose from the **Data event type** list in the CloudTrail console. The **resources.type value** column shows the `resources.type` value, which you would specify when configuring advanced event selectors using the AWS CLI or CloudTrail APIs. The **Data APIs logged to CloudTrail** column shows the API calls logged to CloudTrail for the resource type.

Data event type (console)	resources.type value	Data APIs logged to CloudTrail
AppSync GraphQL	`AWS::AppSync::GraphQLApi`	GraphQL

You can configure advanced event selectors to filter on the `eventName`, `readOnly`, and `resources.ARN` fields to log only those events that are important to you. For more information about these fields, see AdvancedFieldSelector in the *AWS CloudTrail API Reference*.

```
[
  {
    "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"
```

```
        ]
      }
    ]
  }
]
```

Understanding AWS AppSync log file entries

CloudTrail delivers events as log files that contain one or more log entries. An event represents a single request from any source and includes information about the requested operation, the date and time of the operation, the request parameters, and so on. Because these log files aren't an ordered stack trace of the public API calls, they don't appear in any specific order.

> **Note**
>
> The `requestID` isn't an authoritative unique ID for logs emitted from AWS AppSync. The `requestID` can be overwritten by the client. Therefore, you should use caution when making decisions based on this information.

The following example CloudTrail log entry demonstrates the `CreateApiKey` operation.

```
{
  "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"
    }
  ]
}
```

The following example CloudTrail log entry demonstrates the `ListApiKeys` operation.

```
{
  "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"
    }
  ]
}
```

The following example CloudTrail log entry demonstrates the `DeleteApiKey` operation.

```
{
  "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"
    }
  ]
}
```

The following example CloudTrail log entry demonstrates a successful GraphQL mutation authorized with a custom Lambda function authorizer.

```
{
  "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"
}
```

The following example CloudTrail log entry demonstrates a partially successful GraphQL operation authorized with a custom Lambda function authorizer. Note the `fieldAuthorizationResults.deniedFields` property that specifies the denied fields.

```
{
  "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"
}
```

The following example CloudTrail log entry demonstrates a failed GraphQL operation.

```
{
  "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"
}
```

The following example demonstrates a successful GraphQL request.

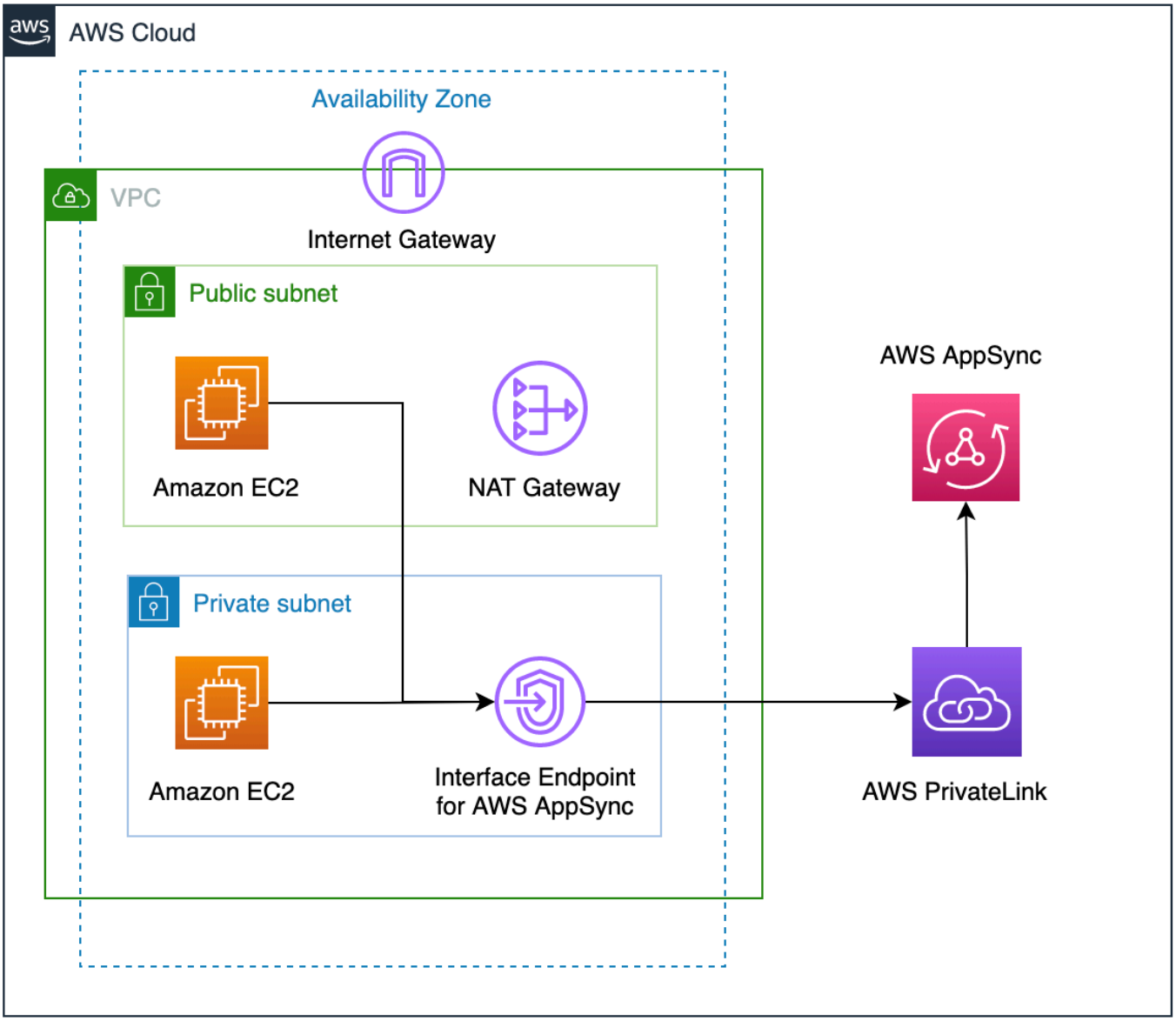
```
{
  "eventVersion": "1.10",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AIDACKCEVSQ6C2EXAMPLE:jane_doe",
    "arn": "arn:aws:sts::123456789012:assumed-role/admin/jane_doe",
    "accountId": "123456789012",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "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"
}
```


Using AWS AppSync Private APIs

If you use Amazon Virtual Private Cloud (Amazon VPC), you can create AWS AppSync Private APIs, which are APIs that can only be accessed from a VPC. With a Private API, you can restrict API access to your internal applications and connect to your GraphQL and Realtime endpoints without exposing data publicly.

To establish a private connection between your VPC and the AWS AppSync service, you must create an interface VPC endpoint. Interface endpoints are powered by AWS PrivateLink, which enables you to privately access AWS AppSync APIs without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to communicate with AWS AppSync APIs. Traffic between your VPC and AWS AppSync doesn't leave the AWS network.



There are some additional factors to consider before enabling Private API features:

- Setting up VPC interface endpoints for AWS AppSync with Private DNS features enabled will prevent resources in the VPC from being able to invoke other AWS AppSync public APIs using the AWS AppSync generated API URL. This is due to the request to the public API being routed via the interface endpoint, which is not allowed for public APIs. To invoke public APIs in this scenario, it is recommended to configure custom domain names on public APIs, which can then be used by resources in the VPC to invoke the public API.

- Your AWS AppSync Private APIs will only be available from your VPC. The AWS AppSync console Query editor will only be able to reach your API if your browser's network configuration can route traffic to your VPC (e.g., connection via VPN or over AWS Direct Connect).
- With a VPC interface endpoint for AWS AppSync, you can access any Private API in the same AWS account and Region. To further restrict access to Private APIs, you can consider the following options:
  - Ensuring only the required administrators can create VPC endpoint interfaces for AWS AppSync.
  - Using VPC endpoint custom policies to restrict which APIs can be invoked from resources in the VPC.
  - For resources in the VPC, we recommend that you use IAM authorization to invoke AWS AppSync APIs by ensuring that the resources are given scoped-down roles to the APIs.
- When creating or using policies that restrict IAM principals, you must set the `authorizationType` of the method to `AWS_IAM` or `NONE`.

Creating AWS AppSync Private APIs

The following steps below show you how to create Private APIs in the AWS AppSync service.

> **Warning**
>
> You can enable Private API features only during the creation of the API. This setting cannot be modified on an AWS AppSync API or an AWS AppSync Private API after it has been created.

1. Sign in to the AWS Management Console and open the AppSync console.
   - In the **Dashboard**, choose **Create API**.
2. Choose **Design an API from scratch**, then choose **Next**.
3. In the **Private API** section, choose **Use Private API features**.
4. Configure the rest of the options, review your API's data, then choose **Create**.

Before you can use your AWS AppSync Private API, you must configure an interface endpoint for AWS AppSync in your VPC. Note that both the Private API and VPC must be in the same AWS account and Region.

Creating an interface endpoint for AWS AppSync

You can create an interface endpoint for AWS AppSync using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see Creating an interface endpoint in the *Amazon VPC User Guide*.

Console

1. Sign in to the AWS Management Console and open the Endpoints page of the Amazon VPC console.
2. Choose **Create endpoint**.
   a. In the **Service category** field, verify that **AWS services** is selected.
   b. In the **Services** table, choose `com.amazonaws.{region}.appsync-api`. Verify that the **Type** column value is `Interface`.
   c. In the **VPC** field, choose a VPC and its subnets.
   d. To enable private DNS features for the interface endpoint, tick the **Enable DNS Name** check box.
   e. In the **Security group** field, choose one or more security groups.
3. Choose **Create endpoint**.

CLI

Use the `create-vpc-endpoint` command and specify the VPC ID, VPC endpoint type (interface), service name, subnets that will use the endpoint, and security groups to associate with the endpoint's network interfaces. For example:

```
$ 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
```

To use the private DNS option, you must set the `enableDnsHostnames` and `enableDnsSupportattributes` values of your VPC. For more information, see Viewing and updating DNS support for your VPC in the *Amazon VPC User Guide*. If you enable private DNS features for the interface endpoint, you can make requests to your AWS AppSync API GraphQL and Real-time endpoint using its default public DNS endpoints using the format below:

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

For more information on service endpoints, see Service endpoints and quotas in the *AWS General Reference*.

For more information on service interactions with interface endpoints, see Accessing a service through an interface endpoint in the *Amazon VPC User Guide*.

For information about creating and configuring an endpoint using AWS CloudFormation, see the AWS::EC2::VPCEndpoint resource in the *AWS CloudFormation User Guide*.

Advanced examples

If you enable private DNS features for the interface endpoint, you can make requests to your AWS AppSync API GraphQL and Real-time endpoint using its default public DNS endpoints using the format below:

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

Using the interface VPC endpoint public DNS hostnames, the base URL to invoke the API will be in the following format:

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.
{region}.vpce.amazonaws.com/graphql
```

You can also use the AZ-specific DNS hostname if you have deployed an endpoint in the AZ:

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}-{az_id}.appsync-api.
{region}.vpce.amazonaws.com/graphql.
```

Using the VPC endpoint public DNS name will require the AWS AppSync API endpoint hostname to be passed as `Host` or as a `x-appsync-domain` header to the request. These examples use a `TodoAPI` that was created in the Launch a sample schema guide:

```
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"}}}'
```

In the following examples, we will use the *Todo* app that is generated in the Launch a sample schema guide. To test out the sample Todo API, we will be using the Private DNS to invoke the API. You can use any command line tool of your choice; this example uses curl to send queries and mutations and wscat to set up subscriptions. To emulate our example, replace the values in brackets { } in the commands below with the corresponding values from your AWS account.

Testing Mutation Operation – `createTodo` Request

```
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"}}}'
```

Testing Mutation Operation – `createTodo` Response

```
{
    "data": {
        "createTodo": {
            "id": "<todo-id>",
            "name": "My first GraphQL task",
            "where": "Day 1",
            "when": "Friday Night",
            "description": "Learn more about GraphQL"
        }
    }
}
```

Testing Query Operation – `listTodos` Request

```
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"}}}'
```

Testing Query Operation – `listTodos` Request

```
{
  "data": {
    "listTodos": {
      "items": [
        {
          "description": "Learn more about GraphQL",
          "id": "<todo-id>",
          "name": "My first GraphQL task",
          "when": "Friday night",
          "where": "Day 1"
        }
      ]
    }
  }
}
```

Testing Subscription Operation – Subscribing to `createTodo` mutation

To set up GraphQL subscriptions in AWS AppSync, see Building a real-time WebSocket client. From an Amazon EC2 instance in a VPC, you can test your AWS AppSync Private API subscription endpoint using wscat. The example below uses an `API KEY` for authorization.

```
$ 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"}
```

Alternatively, use the VPC endpoint domain name while making sure to specify the **Host** header in the `wscat` command to establish the websocket:

```
$ 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"}
```

Run the mutation code below:

```
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"}}}'
```

Afterwards, a subscription is trigged, and the message notification appears as shown below:

```
< {"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"}}}}
```

Using IAM policies to limit public API creation

AWS AppSync supports IAM `Condition` statements for use with Private APIs. The `visibility` field can be included with IAM policy statements for the `appsync:CreateGraphqlApi` operation to control which IAM roles and users can create private and public APIs. This gives an IAM administrator the ability to define an IAM policy that will only allow a user to create a Private GraphQL API. A user attempting to create a public API will receive an unauthorized message.

For example, an IAM administrator could create the following IAM policy statement to allow for the creation of Private APIs:

```
{
    "Sid": "AllowPrivateAppSyncApis",
    "Effect": "Allow",
    "Action": "appsync:CreateGraphqlApi",
    "Resource": "*",
    "Condition": {
        "ForAnyValue:StringEquals": {
            "appsync:Visibility": "PRIVATE"
        }
    }
}
```

An IAM administrator could also add the following service control policy to block all users in an AWS organization from creating AWS AppSync APIs other than Private APIs:

```
{
    "Sid": "BlockNonPrivateAppSyncApis",
    "Effect": "Deny",
    "Action": "appsync:CreateGraphqlApi",
    "Resource": "*",
    "Condition": {
        "ForAnyValue:StringNotEquals": {
            "appsync:Visibility": "PRIVATE"
        }
    }
}
```

Sharing AWS AppSync GraphQL APIs

AWS AppSync integrates with AWS Resource Access Manager (AWS RAM) to enable resource sharing. AWS RAM is a service that enables you to share invoke actions (query, mutation, and subscription operations and connect requests to your real-time WebSocket endpoint) on AWS AppSync GraphQL APIs with other AWS accounts or through AWS Organizations. With AWS RAM, you share resources that you own by creating a resource share. A resource share specifies the resources to share, and the consumers with whom to share them. Consumers can include the following.

- Specific AWS accounts inside or outside of its organization in AWS Organizations
- An organizational unit inside of its organization in AWS Organizations
- An entire organization in AWS Organizations

For more information about AWS RAM, see the AWS Resource Access Manager User Guide.

Topics

Prerequisites for sharing AWS AppSync GraphQL APIs

Sharing AWS AppSync GraphQL APIs has the following prerequisites.

- To share an AWS AppSync GraphQL API, you must own it in your AWS account. This means that the AWS AppSync GraphQL API must be allocated or provisioned in your account.
- To share an AWS AppSync GraphQL API with your organization or an organizational unit in AWS Organizations, you must enable sharing with AWS Organizations. For more information, see Enable resource sharing within AWS Organizations in the *AWS Resource Access Manager User Guide*.

Share AWS AppSync GraphQL APIs

To share an AWS AppSync GraphQL API, start by creating a resource share using AWS Resource Access Manager. A resource share specifies the resources to share, the consumers with whom they are shared, and what actions principals can perform. When you share an AWS AppSync GraphQL API that you own, with other AWS accounts, you enable those accounts to call that AWS AppSync API in your AWS account.

If you are part of an organization in AWS Organizations, and sharing within your organization is enabled, consumers in your organization are automatically granted access to the shared resource. Otherwise, consumers receive an invitation to join the resource share and are granted access to the shared resource after accepting the invitation.

Sharing considerations

- You can share only AWS AppSync GraphQL APIs, not other API types such as Event APIs.
- You can share only AWS AppSync GraphQL APIs that have `AWS_IAM` as one of the authorization modes configured on the API.

  If `AWS_IAM` is removed from the list of authorization modes for a shared AppSync GraphQL API, while the resource share may still exist, it will be rendered ineffective.
- You can share both public and private AWS AppSync GraphQL APIs.
- Private AWS AppSync GraphQL APIs can always be accessed via VPC endpoints in VPCs in the origin AWS account, and all authorization modes are supported, not just `AWS_IAM`.
- For shared AWS AppSync GraphQL APIs, permissions are managed for the API resource only and do not support fine grained permissions for field and type, and field resources. When you share an API, you are sharing the API ARN and the ARNs for all of its types and fields.

Create a resource share that you own using the AWS RAM console

To share an AWS AppSync GraphQL API, use the procedure described in Creating a resource share in the *AWS Resource Access Manager User Guide*, using the RAM permission name `AWSRAMPermissionAppSyncGraphQLApiInvokeAccess`.

Create and use a customer managed permission to share a private AWS AppSync GraphQL API using the AWS RAM console

To share a private AWS AppSync GraphQL API, create a customer managed permission using the procedure described in Creating and using customer managed permissions in the *AWS Resource Access Manager User Guide*.

As an example, an owner for Account A wants to grant principals in Account B permission to access a private AWS AppSync GraphQL API (PrivateApiA) for calls made via VPCE-B (a VPC Endpoint owned by Account B). In this case, the owner for Account A needs to create an AWS RAM customer managed permission as follows.

```
{
    "Effect": "Allow",
    "Action": ["appsync:GraphQL"],
    "Condition": {
        "StringEqualsIgnoreCase": {
            "aws:SourceVpce": [
                "VPCE-B"
            ]
        }
    }
}
```

Assume that this new customer managed AWS RAM permission is named `private-api-A-access-via-vpce-b`.

To enable cross-account access to `PrivateApiA` via `VPCE-B`, the customer can create an AWS RAM resource share with the following parameters and the customer-managed permission in the previous example.

- **Resource Type:** `appsync:Apis`
- **Resource:** `arn:aws:appsync:us-west-2:A:apis/PrivateApiA`
- **Permission:** `private-api-A-access-via-vpce-b` (Customer-managed permission)
- **Principal:** `Account: B`

Create a resource share that you own using the AWS CLI

To share an AWS AppSync GraphQL API using the AWS CLI, use the `create-resource-share` command with `arn:aws:ram::aws:permission/AWSRAMPermissionAppSyncApiInvokeAccess` as the value for the `--permission-arns` switch.

For a complete list of available commands for AWS RAM, see the AWS RAM CLI reference.

Stop sharing AWS AppSync GraphQL APIs

To stop sharing AWS AppSync GraphQL APIs that you own, you must either delete the resource share or update the principals that you shared the resource with. Refer to the documentation in the following sections for the action you want to perform.

To stop sharing a resource that you own using the AWS RAM console

See Update a resource share in the *AWS Resource Access Manager User Guide*.

To stop sharing a resource that you own using the AWS CLI

Use the disassociate-resource-share command.

To delete a resource share you own using the AWS RAM console

See Deleting a resource share in the *AWS Resource Access Manager User Guide*.

To delete a resource share you own using the AWS CLI

Use the delete-resource-share command.

For a complete list of available commands for AWS RAM, see the AWS RAM CLI reference.

Cross-account events

You can opt-in to logging AWS CloudTrail Data Events for monitoring and auditing cross-account AWS AppSync GraphQL API DataPlane activity. For more information, see Logging data events in the *AWS CloudTrail User Guide*.

Configuring GraphQL run complexity, query depth, and introspection with AWS AppSync

AWS AppSync allows you to enable or disable introspection features and set limits to the amount of nested levels and resolvers in a single query.

Using the introspection feature

> **Tip**
>
> For more information about introspection in GraphQL, see this article on the GraphQL foundation's website.

By default, GraphQL allows you to use introspection to query the schema itself to discover its types, fields, queries, mutations, subscriptions, etc. This is an important feature for learning how the data is shaped and processed by your GraphQL service. However, there are some things to consider when dealing with introspection. You may have a use case that would benefit from introspection being disabled, such as a case in which field names may be sensitive or hidden or the full API schema is intended to be left undocumented for consumers. In these cases, publishing schema data through introspection could result in the leakage of intentionally private data.

To prevent this from happening, you can disable introspection. This will prevent unauthorized parties from using introspection fields on your schema. However, it's important to note that introspection is useful for development teams to learn how data in their service is processed. Internally, it might be helpful to keep introspection enabled while disabling it in production code as an extra layer of security. Another way to handle this is to add an authorization method, which AWS AppSync also provides. For more information, see authorization.

AWS AppSync allows you to enable or disable introspection at the API level. To enable or disable introspection, do the following:

1. Sign in to the AWS Management Console and open the AppSync console.
2. On the **APIs** page, choose the name of a GraphQL API.
3. On your API's homepage, in the navigation pane, choose **Settings**.
4. In **API configurations**, choose **Edit**.

5. Under **Introspection queries**, do the following:

   - Turn on or off **Enable introspection queries**.

6. Choose **Save**.

When introspection is enabled (the default behavior), using the introspection system will work normally. For example, the image below shows a `__schema` field processing all available types in the schema:

When disabling this feature, a validation error will appear in the response instead:

Configuring query depth limits

There are times during which you may want more granular control over how the API functions during an operation. One such control is adding a limit to the amount of nested levels a query may process. By default, queries are able to process an unlimited amount of nested levels. Limiting queries to a specified amount of nested levels has potential implications for the performance and flexibility of your project. Take the following query:

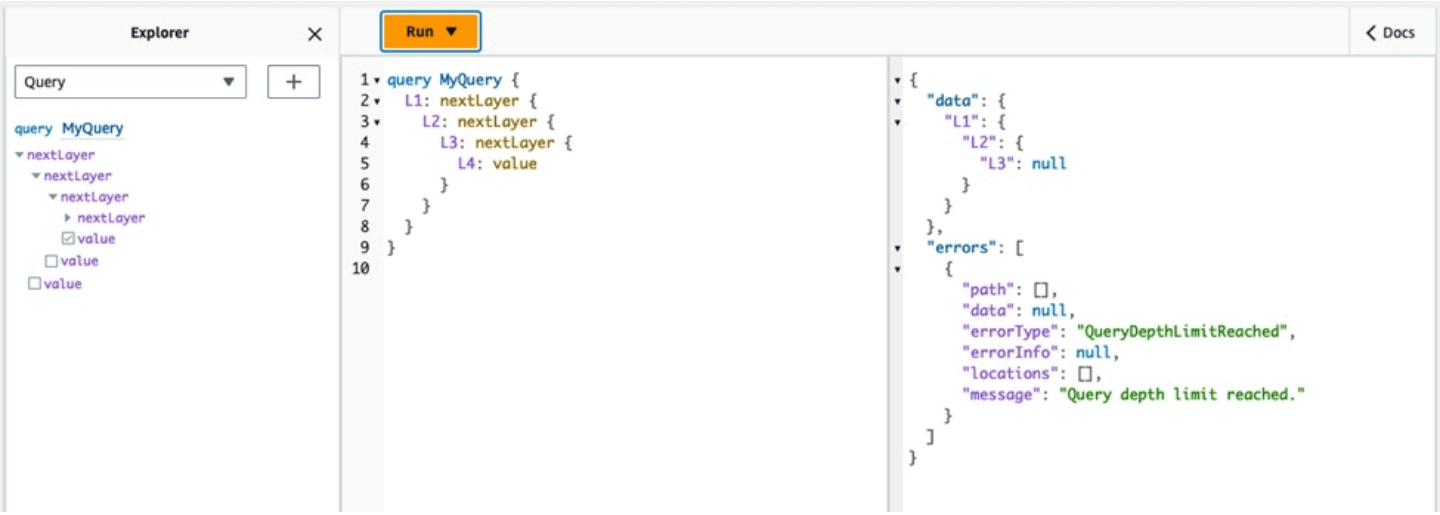
```
query MyQuery {
  L1: nextLayer {
    L2: nextLayer {
      L3: nextLayer {
        L4: value
      }
    }
  }
}
```

Your project may call for limiting queries to L1 or L2 for some purpose. By default, the entire query from L1 to L4 would be processed with no way to control that. By setting a limit, you could prevent queries from accessing anything past the specified level.

To add a query depth limit, do the following:

1. Sign in to the AWS Management Console and open the AppSync console.
2. On the **APIs** page, choose the name of a GraphQL API.
3. On your API's homepage, in the navigation pane, choose **Settings**.
4. In **API configurations**, choose **Edit**.
5. Under **Query depth**, do the following:
    a. Turn on or off **Enable query depth**.
    b. In **Maximum depth**, set the depth limit. This can be between 1 and 75.
6. Choose **Save**.

When a limit is set, going past its upper bound will result in a `QueryDepthLimitReached` error. For example, the image below shows a query with a depth limit of 2 going past the limit to the third (L3) and fourth (L4) levels:



Note that fields can still be marked as nullable or non-nullable in the schema. If a non-nullable field receives a `QueryDepthLimitReached` error, that error will be thrown to the first nullable parent field.

Configuring resolver count limits

You can also control how many resolvers each query can process. Like the query depth, you can set a limit to this amount. Take the following query that contains three resolvers:

```
query MyQuery {
  resolver1: resolver
  resolver2: resolver
  resolver3: resolver
}
```

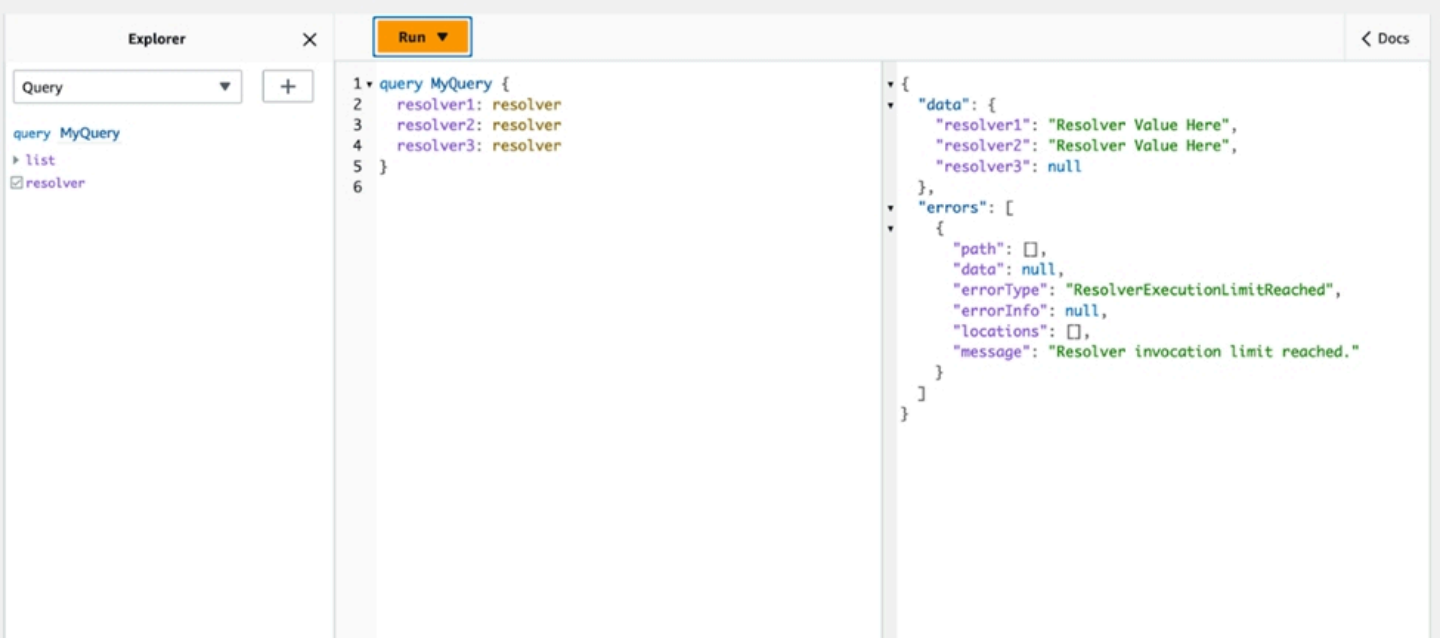
By default, each query can process up to 10000 resolvers. In the example above, `resolver1`, `resolver2`, and `resolver3` will be processed. However, your project may call for limiting each query to handling one or two resolvers in total. By setting a limit, you can tell the query to not handle any resolver past a certain number like the first (`resolver1`) or second (`resolver2`) resolvers.

To add a resolver count limit, do the following:

1. Sign in to the AWS Management Console and open the AppSync console.
2. On the **APIs** page, choose the name of a GraphQL API.
3. On your API's homepage, in the navigation pane, choose **Settings**.

4. In **API configurations**, choose **Edit**.
5. Under **Resolver count limit**, do the following:
   a. Turn on **Enable resolver count**.
   b. In **Maximum resolver count**, set the count limit. This can be between 1 and 10000.
6. Choose **Save**.

Like the query depth limit, exceeding the configured resolver limit causes the query to end with a `ResolverExecutionLimitReached` error on additional resolvers. In the image below, a query with a resolver count limit of *2* tries to process three resolvers. Because of the limit, the third resolver throws an error and doesn't run.



Using environment variables in AWS AppSync

You can use environment variables to adjust your AWS AppSync resolvers' and functions' behavior without updating your code. Environment variables are pairs of strings stored with your API configuration that are made available to your resolvers and functions to leverage at runtime. They're particularly useful for situations in which you must reference configuration data that's only available during the initial setup but needs to be used by your resolvers and functions during the run. Environment variables expose configuration data in your code, thereby reducing the need to hard-code those values.

> **Note**
>
> To increase database security, we recommend that you use Secrets Manager or AWS Systems Manager Parameter Store instead of environment variables to store credentials or sensitive information. To leverage this feature, see Invoking AWS services with AWS AppSync HTTP data sources.

Environment variables must follow several behaviors and rules to function properly:

- Both JavaScript resolvers/functions and VTL templates support environment variables.
- Environment variables are not evaluated before function invocation.
- Environment variables **only** support string values.
- Any defined value in an environment variable is considered a string literal and not expanded.
- Variable evaluations should ideally be performed in the function code.

Configuring environment variables (console)

You can configure environment variables for your AWS AppSync GraphQL API by creating the variable and defining its key-value pair. Your resolvers and functions will use the environment variable's key name to retrieve the value at runtime. To set environment variables in the AWS AppSync console:

1. Sign in to the AWS Management Console and open the AppSync console.
2. On the **APIs** page, choose the name of a GraphQL API.
3. On your API's homepage, in the navigation pane, choose **Settings**.
4. Under **Environment variables**, choose **Add environment variable**.
5. Choose **Add environment variable**.
6. Enter a key and value.
7. If necessary, repeat steps 5 and 6 to add more key values. If you need to remove a key value, choose the **Remove** option and the key(s) to remove.
8. Choose **Submit**.

Tip

There are a few rules you must follow when creating keys and values:

- Keys must begin with a letter.
- Keys must be at least two characters long.
- Keys can only contain letters, numbers, and the underscore character (_).
- Values can be up to 512 characters long.
- You can configure up to 50 key-value pairs in a GraphQL API.

Configuring environment variables (API)

To set an environment variable using APIs, you can use `PutGraphqlApiEnvironmentVariables`. The corresponding CLI command is `put-graphql-api-environment-variables`.

To retrieve an environment variable using APIs, you can use `GetGraphqlApiEnvironmentVariables`. The corresponding CLI command is `get-graphql-api-environment-variables`.

The command must contain the API ID and list of environment variables:

```
aws appsync put-graphql-api-environment-variables \
  --api-id "<api-id>" \
  --environment-variables '{"key1":"value1","key2":"value2", …}'
```

The following example sets two environment variables in an API with the ID of `abcdefghijklmnopqrstuvwxyz` using the `put-graphql-api-environment-variables` command:

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true"}'
```

Note that when you apply environment variables with the `put-graphql-api-environment-variables` command, the contents of the environment variables' structure are overwritten; this means existing environment variables will be lost. To retain existing environment variables when

adding new ones, **include all existing key-value pairs** along with the new ones in your request. Using the example above, if you wanted to add `"EMPTY":""`, you could do the following:

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true", "EMPTY":""}'
```

To retrieve the current configuration, use the `get-graphql-api-environment-variables` command:

```
aws appsync get-graphql-api-environment-variables --api-id "<api-id>"
```

Using the example above, you could use the following command:

```
aws appsync get-graphql-api-environment-variables --api-id "abcdefghijklmnopqrstuvwxyz"
```

The result will show the list of environment variables along with their key values:

```
{
    "environmentVariables": {
        "USER_TABLE": "users_prod",
        "DEBUG": "true",
        "EMPTY": ""
    }
}
```

Configuring environment variables (CFN)

You can use the template below to create environment variables:

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  GraphQLApiWithEnvVariables:
    Type: "AWS::AppSync::GraphQLApi"
    Properties:
      Name: "MyApiWithEnvVars"
      AuthenticationType: "AWS_IAM"
      EnvironmentVariables:
        EnvKey1: "non-empty"
        EnvKey2: ""
```

environment variables and merged APIs

Environment variables defined in Source APIs are also available in your Merged APIs. Environment variables in Merged APIs are read-only and cannot be updated. Note that your environment variable keys must be unique across all Source APIs for your merges to succeed; duplicate keys will always result in a merge failure.

Retrieving environment variables

To retrieve environment variables in your function code, retrieve the value from the `ctx.env` object in your resolvers and functions. Below are some examples of this in action.

Publishing to Amazon SNS

In this example, our HTTP resolver sends a message to an Amazon SNS topic. The ARN of the topic is only known after the stack that defines the GraphQL API and the topic has been deployed.

```
/**
 * 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

In this example, the names of the DynamoDB table are different if the API is deployed for staging or is already in production. The resolver code doesn't need to change. The values of the environment variables are updated based on where the API is deployed.

```
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
      },
    ],
  };
}
```

Configuring authorization and authentication to secure your GraphQL APIs

AWS AppSync offers the following authorization types to secure GraphQL APIs: API keys, Lambda, IAM, OpenID Connect, and Cognito User Pools. Each option provides a different method of security:

1. **API Key Authorization**: Controls throttling for unauthenticated APIs, providing a simple security option.
2. **Lambda Authorization**: Enables custom authorization logic, explaining function inputs and outputs in detail.
3. **IAM Authorization**: Utilizes AWS's signature version 4 signing process, allowing fine-grained access control through IAM policies.
4. **OpenID Connect Authorization**: Integrates with OIDC-compliant services for user authentication.
5. **Cognito User Pools**: Implements group-based access control using Cognito's user management features.

Authorization types

There are five ways you can authorize applications to interact with your AWS AppSync GraphQL API. You specify which authorization type you use by specifying one of the following authorization type values in your AWS AppSync API or CLI call:

- **`API_KEY`**

  For using API keys.
- **`AWS_LAMBDA`**

  For using an AWS Lambda function.
- **`AWS_IAM`**

  For using AWS Identity and Access Management (IAM) permissions.
- **`OPENID_CONNECT`**

  For using your OpenID Connect provider.

- **`AMAZON_COGNITO_USER_POOLS`**

  For using an Amazon Cognito user pool.

These basic authorization types work for most developers. For more advanced use cases, you can add additional authorization modes through the console, the CLI, and AWS CloudFormation. For additional authorization modes, AWS AppSync provides an authorization type that takes the values listed above (that is, `API_KEY`, `AWS_LAMBDA`, `AWS_IAM`, `OPENID_CONNECT`, and `AMAZON_COGNITO_USER_POOLS`).

When you specify `API_KEY`,`AWS_LAMBDA`, or `AWS_IAM` as the main or default authorization type, you can't specify them again as one of the additional authorization modes. Similarly, you can't duplicate `API_KEY`, `AWS_LAMBDA` or `AWS_IAM` inside the additional authorization modes. You can use multiple Amazon Cognito User Pools and OpenID Connect providers. However, you can't use duplicate Amazon Cognito User Pools or OpenID Connect providers between the default authorization mode and any of the additional authorization modes. You can specify different clients for your Amazon Cognito User Pool or OpenID Connect provider using the corresponding configuration regular expression.

When you save changes to your API configuration, AWS AppSync starts to propagate the changes. Until your configuration change is propagated, AWS AppSync continues to serve your content from the previous configuration. After your configuration change is propagated, AWS AppSync immediately starts to serve your content based on the new configuration. While AWS AppSync is propagating your changes for an API, we can't determine whether the API is serving your content based on the previous configuration or the new configuration.

API_KEY authorization

Unauthenticated APIs require more strict throttling than authenticated APIs. One way to control throttling for unauthenticated GraphQL endpoints is through the use of API keys. An API key is a hard-coded value in your application that is generated by the AWS AppSync service when you create an unauthenticated GraphQL endpoint. You can rotate API keys from the console, from the CLI, or from the AWS AppSync API reference.

Console

1. Sign in to the AWS Management Console and open the AppSync console.

   a. In the **APIs dashboard**, choose your GraphQL API.

b. In the **Sidebar**, choose **Settings**.

2. Under **Default authorization mode**, choose **API key**.
3. In the **API keys** table, choose **Add API key**.

   A new API key will be generated in the table.

   - To delete an old API key, select the API key in the table and then choose **Delete**.
4. Choose **Save** at the bottom of the page.

CLI

1. If you haven't already done so, configure your access to the AWS CLI. For more information, see Configuration basics.
2. Create a GraphQL API object by running the `update-graphql-api` command.

   You'll need to type in two parameters for this particular command:

   1. The `api-id` of your GraphQL API.
   2. The new `name` of your API. You can use the same `name`.
   3. The `authentication-type`, which will be `API_KEY`.

   > **Note**
   >
   > There are other parameters such as `Region` that must be configured but will usually default to your CLI configuration values.

   An example command may look like this:

```
aws appsync update-graphql-api --api-id abcdefghijklmnopqrstuvwxyz --name
 TestAPI --authentication-type API_KEY
```

   An output will be returned in the CLI. Here's an example in JSON:

```
{
    "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 keys are configurable for up to 365 days, and you can extend an existing expiration date for up to another 365 days from that day. API Keys are recommended for development purposes or use cases where it's safe to expose a public API.

On the client, the API key is specified by the header `x-api-key`.

For example, if your `API_KEY` is `'ABC123'`, you can send a GraphQL query via `curl` as follows:

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "x-api-key:ABC123" -d
 '{ "query": "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

AWS_LAMBDA authorization

You can implement your own API authorization logic using an AWS Lambda function. You can use a Lambda function for either your primary or secondary authorizer, but there may only be one Lambda authorization function per API. When using Lambda functions for authorization, the following applies:

- If the API has the `AWS_LAMBDA` and `AWS_IAM` authorization modes enabled, then the SigV4 signature cannot be used as the `AWS_LAMBDA` authorization token.
- If the API has the `AWS_LAMBDA` and `OPENID_CONNECT` authorization modes or the `AMAZON_COGNITO_USER_POOLS` authorization mode enabled, then the OIDC token cannot be used as the `AWS_LAMBDA` authorization token. Note that the OIDC token can be a Bearer scheme.
- A Lambda function must not return more than 5MB of contextual data for resolvers.

For example, if your authorization token is `'ABC123'`, you can send a GraphQL query via curl as follows:

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "Authorization:ABC123" -d
 '{ "query":
        "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

Lambda functions are called before each query or mutation. The return value can be cached based on the API ID and the authentication token. When a Lambda authorizer response is less than 1,048,576 bytes, AWS AppSync caches the response for subsequent requests. If the Lambda authorizer response is equal to or greater than 1,048,576 bytes, AWS AppSync doesn't cache the response and invokes the Lambda authorizer for each incoming request. To optimize performance and minimize Lambda invocation costs, we recommend that you limit your Lambda authorizer responses to 1,048,576 bytes. By default, caching is not turned on, but this can be enabled at the API level or by setting the `ttlOverride` value in a function's return value.

A regular expression that validates authorization tokens before the function is called can be specified if desired. These regular expressions are used to validate that an authorization token is of the correct format before your function is called. Any request using a token which does not match this regular expression will be denied automatically.

Lambda functions used for authorization require a principal policy for `appsync.amazonaws.com` to be applied on them to allow AWS AppSync to call them. This action is done automatically in the AWS AppSync console; The AWS AppSync console does *not* remove the policy. For more information on attaching policies to Lambda functions, see Resource-based policies in the AWS Lambda Developer Guide.

The Lambda function you specify will receive an event with the following shape:

```
{
    "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
    }
}
```

The `event` object contains the headers that were sent in the request from the application client to AWS AppSync.

The authorization function must return at least `isAuthorized`, a boolean indicating if the request is authorized. AWS AppSync recognizes the following keys returned from Lambda authorization functions:

> **Note**
>
> The value for the `operationName` in the `requestContext` for a WebSocket connect operation is set by AWS AppSync to "`DeepDish:Connect`".

Functions list

`isAuthorized` (boolean, required)

A boolean value indicating if the value in `authorizationToken` is authorized to make calls to the GraphQL API.

If this value is true, execution of the GraphQL API continues. If this value is false, an `UnauthorizedException` is raised

`deniedFields` (list of string, optional)

A list of which are forcibly changed to `null`, even if a value was returned from a resolver.

Each item is either a fully qualified field ARN in the form of `arn:aws:appsync:`*`us-east-1`*`:`*`111122223333`*`:apis/`*`GraphQLApiId`*`/types/`*`TypeName`*`/fields/`*`FieldName` or a short form of *`TypeName`*`.`*`FieldName`*. The full ARN form should be used when two APIs share a Lambda function authorizer and there might be ambiguity between common types and fields between the two APIs.

`resolverContext` (JSON Object, optional)

A JSON object visible as `$ctx.identity.resolverContext` in resolver templates. For example, if the following structure is returned by a resolver:

```
{
  "isAuthorized":true
  "resolverContext": {
    "banana":"very yellow",
    "apple":"very green"
  }
}
```

The value of `ctx.identity.resolverContext.apple` in resolver templates will be "`very green`". The `resolverContext` object only supports key-value pairs. Nested keys are not supported.

> **⚠ Warning**
>
> The total size of this JSON object must not exceed 5MB.

`ttlOverride` (integer, optional)

The number of seconds that the response should be cached for. If no value is returned, the value from the API is used. If this is 0, the response is not cached.

Lambda authorizers have a timeout of 10 seconds. We recommend designing functions to execute in the shortest amount of time as possible to scale the performance of your API.

Multiple AWS AppSync APIs can share a single authentication Lambda function. Cross account authorizer use is not permitted.

When sharing an authorization function between multiple APIs, be aware that short-form field names (*`typename`*`.`*`fieldname`*) may inadvertently hide fields. To disambiguate a field in `deniedFields`, you can specify an unambiguous field ARN in the form of `arn:aws:appsync:`*`region`*`:`*`accountId`*`:apis/`*`GraphQLApiId`*`/types/`*`typeName`*`/fields/`*`fieldName`*.

To add a Lambda function as the default authorization mode in AWS AppSync:

Console

1. Log into the AWS AppSync Console and navigate to the API you wish to update.
2. Navigate to the Settings page for your API.

Change the API-Level authorization to **AWS Lambda**.

3. Choose the AWS Region and Lambda ARN to authorize API calls against.

> **Note**
>
> The appropriate principal policy will be added automatically, allowing AWS AppSync to call your Lambda function.

4. Optionally, set the response TTL and token validation regular expression.

AWS CLI

1. Attach the following policy to the Lambda function being used:

```
aws lambda add-permission --function-name "my-function" --statement-id "appsync"
 --principal appsync.amazonaws.com --action lambda:InvokeFunction --output text
```

> **Important**
>
> If you want the policy of the function to be locked to a single GraphQL API, you can run this command:
>
> ```
> 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. Update your AWS AppSync API to use the given Lambda function ARN as the authorizer:

```
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**
>
> You can also include other configuration options such as the token regular expression.

The following example describes a Lambda function that demonstrates the various authentication and failure states a Lambda function can have when used as a AWS AppSync authorization mechanism:

```
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 {}
```

Circumventing SigV4 and OIDC token authorization limitations

The following methods can be used to circumvent the issue of not being able to use your SigV4 signature or OIDC token as your Lambda authorization token when certain authorization modes are enabled.

If you want to use the SigV4 signature as the Lambda authorization token when the `AWS_IAM` and `AWS_LAMBDA` authorization modes are enabled for AWS AppSync's API, do the following:

- To create a new Lambda authorization token, add random suffixes and/or prefixes to the SigV4 signature.
- To retrieve the original SigV4 signature, update your Lambda function by removing the random prefixes and/or suffixes from the Lambda authorization token. Then, use the original SigV4 signature for authentication.

If you want to use the OIDC token as the Lambda authorization token when the `OPENID_CONNECT` authorization mode or the `AMAZON_COGNITO_USER_POOLS` and `AWS_LAMBDA` authorization modes are enabled for AWS AppSync's API, do the following:

- To create a new Lambda authorization token, add random suffixes and/or prefixes to the OIDC token. The Lambda authorization token should not contain a Bearer scheme prefix.
- To retrieve the original OIDC token, update your Lambda function by removing the random prefixes and/or suffixes from the Lambda authorization token. Then, use the original OIDC token for authentication.

AWS_IAM authorization

This authorization type enforces the AWS signature version 4 signing process on the GraphQL API. You can associate Identity and Access Management (IAM) access policies with this authorization type. Your application can leverage this association by using an access key (which consists of an access key ID and secret access key) or by using short-lived, temporary credentials provided by Amazon Cognito Federated Identities.

If you want a role that has access to perform all data operations:

JSON

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "appsync:GraphQL"
         ],
         "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/*"
         ]
      }
   ]
}
```

You can find `YourGraphQLApiId` from the main API listing page in the AppSync console, directly under the name of your API. Alternatively you can retrieve it with the CLI: `aws appsync list-graphql-apis`

If you want to restrict access to just certain GraphQL operations, you can do this for the root `Query`, `Mutation`, and `Subscription` fields.

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>"
         ]
     }
   ]
}
```

For example, suppose you have the following schema and you want to restrict access to getting all posts:

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

type Query {
   posts:[Post!]!
}

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

The corresponding IAM policy for a role (that you could attach to an Amazon Cognito identity pool, for example) would look like the following:

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 authorization

This authorization type enforces OpenID connect (OIDC) tokens provided by an OIDC-compliant service. Your application can leverage users and privileges defined by your OIDC provider for controlling access.

An Issuer URL is the only required configuration value that you provide to AWS AppSync (for example, `https://auth.example.com`). This URL must be addressable over HTTPS. AWS AppSync appends `/.well-known/openid-configuration` to the issuer URL and locates the OpenID configuration at `https://auth.example.com/.well-known/openid-configuration` per the OpenID Connect Discovery specification. It expects to retrieve an RFC5785 compliant JSON document at this URL. This JSON document must contain a `jwks_uri` key, which points to the JSON Web Key Set (JWKS) document with the signing keys. AWS AppSync requires the JWKS to contain JSON fields of `kty` and `kid`.

AWS AppSync supports a wide range of signing algorithms.

Signing algorithms
RS256
RS384
RS512
PS256
PS384
PS512
HS256
HS384

Signing algorithms
HS512
ES256
ES384
ES512

We recommend that you use the RSA algorithms. Tokens issued by the provider must include the time at which the token was issued (`iat`) and may include the time at which it was authenticated (`auth_time`). You can provide TTL values for issued time (`iatTTL`) and authentication time (`authTTL`) in your OpenID Connect configuration for additional validation. If your provider authorizes multiple applications, you can also provide a regular expression (`clientId`) that is used to authorize by client ID. When the `clientId` is present in your OpenID Connect configuration, AWS AppSync validates the claim by requiring the `clientId` to match with either the `aud` or `azp` claim in the token.

To validate multiple client IDs use the pipeline operator ("|") which is an "or" in regular expression. For example, if your OIDC application has four clients with client IDs such as 0A1S2D, 1F4G9H, 1J6L4B, 6GS5MG, to validate only the first three client IDs, you would place 1F4G9H|1J6L4B|6GS5MG in the client ID field.

If an API is configured with multiple authorization types, AWS AppSync validates the issuer (iss claim) present in the JWT token from request headers by comparing it against the issuer URL specified in the API configuration. However, when an API is configured with only OPENID_CONNECT authorization, AWS AppSync skips this issuer URL validation step.

AMAZON_COGNITO_USER_POOLS authorization

This authorization type enforces OIDC tokens provided by Amazon Cognito User Pools. Your application can leverage the users and groups in both your user pools and user pools from another AWS account and associate these with GraphQL fields for controlling access.

When using Amazon Cognito User Pools, you can create groups that users belong to. This information is encoded in a JWT token that your application sends to AWS AppSync in an authorization header when sending GraphQL operations. You can use GraphQL directives on

the schema to control which groups can invoke which resolvers on a field, thereby giving more controlled access to your customers.

For example, suppose you have the following GraphQL schema:

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

type Query {
   posts:[Post!]!
}

type Mutation {
   addPost(id:ID!, title:String!):Post!
}
...
```

If you have two groups in Amazon Cognito User Pools - bloggers and readers - and you want to restrict the readers so that they cannot add new entries, then your schema should look like this:

```
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"])
}
...
```

Note that you can omit the `@aws_auth` directive if you want to default to a specific grant-or-deny strategy on access. You can specify the grant-or-deny strategy in the user pool configuration when you create your GraphQL API via the console or via the following CLI command:

```
$ 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"}'
```

Using additional authorization modes

When you add additional authorization modes, you can directly configure the authorization setting at the AWS AppSync GraphQL API level (that is, the `authenticationType` field that you can directly configure on the `GraphqlApi` object) and it acts as the default on the schema. This means that any type that doesn't have a specific directive has to pass the API level authorization setting.

At the schema level, you can specify additional authorization modes using directives on the schema. You can specify authorization modes on individual fields in the schema. For example, for `API_KEY` authorization you would use `@aws_api_key` on schema object type definitions/fields. The following directives are supported on schema fields and object type definitions:

- `@aws_api_key` - To specify the field is `API_KEY` authorized.
- `@aws_iam` - To specify that the field is `AWS_IAM` authorized.
- `@aws_oidc` - To specify that the field is `OPENID_CONNECT` authorized.
- `@aws_cognito_user_pools` - To specify that the field is `AMAZON_COGNITO_USER_POOLS` authorized.
- `@aws_lambda` - To specify that the field is `AWS_LAMBDA` authorized.

You can't use the `@aws_auth` directive along with additional authorization modes. `@aws_auth` works only in the context of `AMAZON_COGNITO_USER_POOLS` authorization with no additional authorization modes. However, you can use the `@aws_cognito_user_pools` directive in place of the `@aws_auth` directive, using the same arguments. The main difference between the two is that you can specify `@aws_cognito_user_pools` on any field and object type definitions.

To understand how the additional authorization modes work and how they can be specified on a schema, let's have a look at the following schema:

```
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!
}
...
```

For this schema, assume that `AWS_IAM` is the default authorization type on the AWS AppSync GraphQL API. This means that fields that don't have a directive are protected using `AWS_IAM`. For example, that's the case for the `getPost` field on the `Query` type. Schema directives enable you to use more than one authorization mode. For example, you can have `API_KEY` configured as an additional authorization mode on the AWS AppSync GraphQL API, and you can mark a field using the `@aws_api_key` directive (for example, `getAllPosts` in this example). Directives work at the field level so you need to give `API_KEY` access to the `Post` type too. You can do this either by marking each field in the `Post` type with a directive, or by marking the `Post` type with the `@aws_api_key` directive.

To further restrict access to fields in the `Post` type you can use directives against individual fields in the `Post` type as shown following.

For example, you can add a `restrictedContent` field to the `Post` type and restrict access to it by using the `@aws_iam` directive. `AWS_IAM` authenticated requests could access `restrictedContent`, however, `API_KEY` requests wouldn't be able to access it.

```
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
}
...
```

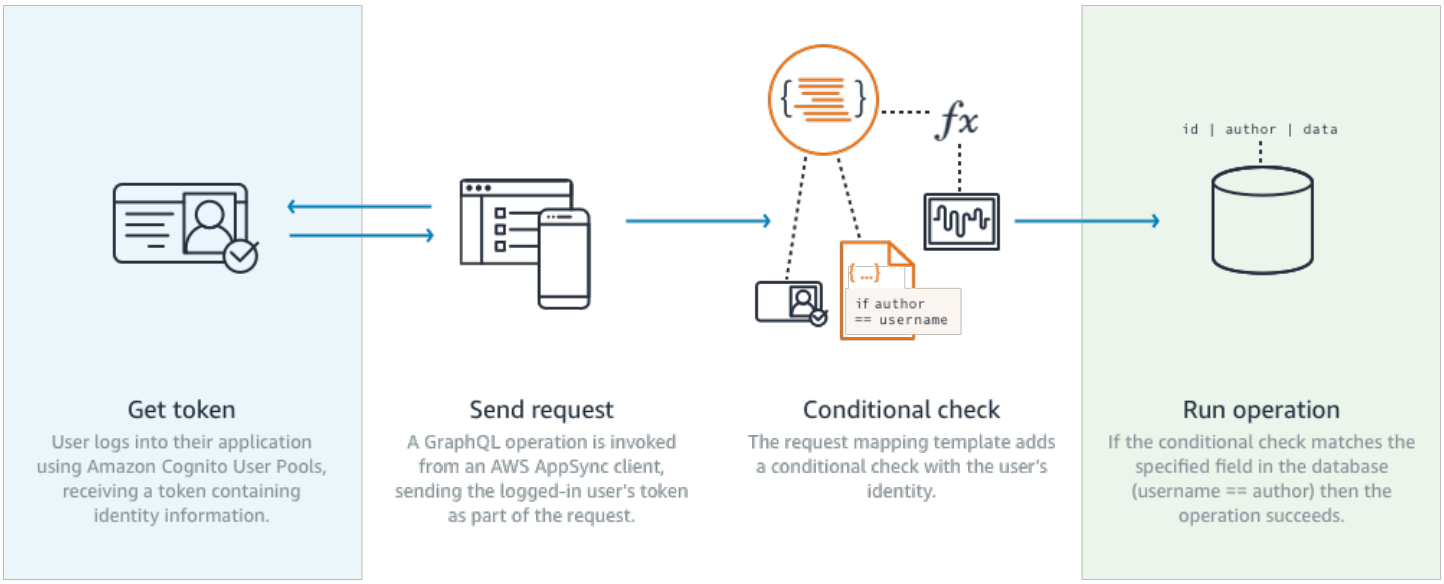
Fine-grained access control

The preceding information demonstrates how to restrict or grant access to certain GraphQL fields. If you want to set access controls on the data based on certain conditions (for example, based on the user that's making a call and whether the user owns the data) you can use mapping templates in your resolvers. You can also perform more complex business logic, which we describe in Filtering Information.

This section shows how to set access controls on your data using a DynamoDB resolver mapping template.

Before proceeding any further, if you're not familiar with mapping templates in AWS AppSync, you may want to review the Resolver mapping template reference and the Resolver mapping template reference for DynamoDB.

In the following example using DynamoDB, suppose you're using the preceding blog post schema, and only users that created a post are allowed to edit it. The evaluation process would be for the user to gain credentials in their application, using Amazon Cognito User Pools for example, and then pass these credentials as part of a GraphQL operation. The mapping template will then substitute a value from the credentials (like the username)in a conditional statement which will then be compared to a value in your database.



To add this functionality, add a GraphQL field of `editPost` as follows:

```
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!
}
...
```

The resolver mapping template for `editPost` (shown in an example at the end of this section) needs to perform a logical check against your data store to allow only the user that created a post to edit it. Since this is an edit operation, it corresponds to an `UpdateItem` in DynamoDB. You can perform a conditional check before performing this action, using context passed through for user identity validation. This is stored in an `Identity` object that has the following values:

```
{
   "accountId" : "12321434323",
   "cognitoIdentityPoolId" : "",
   "cognitoIdentityId" : "",
```

```
   "sourceIP" : "",
   "caller" : "ThisistheprincipalARN",
   "username" : "username",
   "userArn" : "Sameasabove"
}
```

To use this object in a DynamoDBUpdateItem call, you need to store the user identity information in the table for comparison. First, your addPost mutation needs to store the creator. Second, your editPost mutation needs to perform the conditional check before updating.

Here is an example of the resolver code for addPost that stores the user identity as an Author column:

```
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;
```

Note that the Author attribute is populated from the Identity object, which came from the application.

Finally, here is an example of the resolver code for editPost, which only updates the content of the blog post if the request comes from the user that created the post:

```
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;
```

This example uses a `PutItem` that overwrites all values rather than an `UpdateItem`, but the same concept applies on the `condition` statement block.

Filtering information

There may be cases where you cannot control the response from your data source, but you don't want to send unnecessary information to clients on a successful write or read to the data source. In these cases, you can filter information by using a response mapping template.

For example, suppose you don't have an appropriate index on your blog post DynamoDB table (such as an index on `Author`). You could use the following resolver:

```
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;
}
```

The request handler fetches the item even if the caller isn't the author who created the post. To prevent this from returning all data, the response handler checks to make sure the caller matches the item's author. If the caller doesn't match this check, only a null response is returned.

Data source access

AWS AppSync communicates with data sources using Identity and Access Management (IAM) roles and access policies. If you are using an existing role, a Trust Policy needs to be added in order for AWS AppSync to assume the role. The trust relationship will look like below:

JSON

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

It's important to scope down the access policy on the role to only have permissions to act on the minimal set of resources necessary. When using the AppSync console to create a data source and create a role, this is done automatically for you. However when using a built in sample template from the IAM console to create a role outside of the AWS AppSync console the permissions will not be automatically scoped down on a resource and you should perform this action before moving your application to production.

Access control use cases for securing requests and responses

In the Security section you learned about the different Authorization modes for protecting your API and an introduction was given on Fine Grained Authorization mechanisms to understand the concepts and flow. Since AWS AppSync allows you to perform logic full operations on data through the use of GraphQL Resolver Mapping templates, you can protect data on read or write in a very flexible manner using a combination of user identity, conditionals, and data injection.

If you're not familiar with editing AWS AppSync Resolvers, review the programming guide.

Overview

Granting access to data in a system is traditionally done through an Access control matrix where the intersection of a row (resource) and column (user/role) is the permissions granted.

AWS AppSync uses resources in your own account and threads identity (user/role) information into the GraphQL request and response as a context object, which you can use in the resolver. This

means that permissions can be granted appropriately either on write or read operations based on the resolver logic. If this logic is at the resource level, for example only certain named users or groups can read/write to a specific database row, then that "authorization metadata" must be stored. AWS AppSync does not store any data so therefore you must store this authorization metadata with the resources so that permissions can be calculated. Authorization metadata is usually an attribute (column) in a DynamoDB table, such as an **owner** or list of users/groups. For example there could be **Readers** and **Writers** attributes.

From a high level, what this means is that if you are reading an individual item from a data source, you perform a conditional `#if () ... #end` statement in the response template after the resolver has read from the data source. The check will normally be using user or group values in `$context.identity` for membership checks against the authorization metadata returned from a read operation. For multiple records, such as lists returned from a table `Scan` or `Query`, you'll send the condition check as part of the operation to the data source using similar user or group values.

Similarly when writing data you'll apply a conditional statement to the action (like a `PutItem` or `UpdateItem` to see if the user or group making a mutation has permission. The conditional again will many times be using a value in `$context.identity` to compare against authorization metadata on that resource. For both request and response templates you can also use custom headers from clients to perform validation checks.

Reading data

As outlined above the authorization metadata to perform a check must be stored with a resource or passed in to the GraphQL request (identity, header, etc.). To demonstrate this suppose you have the DynamoDB table below:

ID	Data	PeopleCanAccess	GroupsCanAccess	Owner
123	{my: data,...}	[Mary, Joe]	[Admins, Editors]	Nadia

The primary key is `id` and the data to be accessed is `Data`. The other columns are examples of checks you can perform for authorization. `Owner` would be a `String` while `PeopleCanAccess` and `GroupsCanAccess` would be `String Sets` as outlined in the Resolver mapping template reference for DynamoDB.

In the resolver mapping template overview the diagram shows how the response template contains not only the context object but also the results from the data source. For GraphQL queries

of individual items, you can use the response template to check if the user is allowed to see these results or return an authorization error message. This is sometimes referred to as an "Authorization filter". For GraphQL queries returning lists, using a Scan or Query, it is more performant to perform the check on the request template and return data only if an authorization condition is satisfied. The implementation is then:

1. GetItem - authorization check for individual records. Done using `#if() ... #end` statements.
2. Scan/Query operations - authorization check is a `"filter":{"expression":...}` statement. Common checks are equality (`attribute = :input`) or checking if a value is in a list (`contains(attribute, :input)`).

In #2 the `attribute` in both statements represents the column name of the record in a table, such as `Owner` in our above example. You can alias this with a # sign and use `"expressionNames": {...}` but it's not mandatory. The `:input` is a reference to the value you're comparing to the database attribute, which you will define in `"expressionValues":{...}`. You'll see these examples below.

Use case: owner can read

Using the table above, if you only wanted to return data if `Owner == Nadia` for an individual read operation (`GetItem`) your template would look like:

```
#if($context.result["Owner"] == $context.identity.username)
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

A couple things to mention here which will be re-used in the remaining sections. First, the check uses `$context.identity.username` which will be the friendly user sign-up name if Amazon Cognito user pools is used and will be the user identity if IAM is used (including Amazon Cognito Federated Identities). There are other values to store for an owner such as the unique "Amazon Cognito identity" value, which is useful when federating logins from multiple locations, and you should review the options available in the Resolver Mapping Template Context Reference.

Second, the conditional else check responding with `$util.unauthorized()` is completely optional but recommended as a best practice when designing your GraphQL API.

Use case: hardcode specific access

```
// 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
```

Use case: filtering a list of results

In the previous example you were able to perform a check against `$context.result` directly as it returned a single item, however some operations like a scan will return multiple items in `$context.result.items` where you need to perform the authorization filter and only return results that the user is allowed to see. Suppose the `Owner` field had the Amazon Cognito IdentityID this time set on the record, you could then use the following response mapping template to filter to only show those records that the user owned:

```
#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)
```

Use case: multiple people can read

Another popular authorization option is to allow a group of people to be able to read data. In the example below the `"filter":{"expression":...}` only returns values from a table scan if the user running the GraphQL query is listed in the set for `PeopleCanAccess`.

```
{
    "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)
        }
    }
}
```

Use case: group can read

Similar to the last use case, it may be that only people in one or more groups have rights to read certain items in a database. Use of the `"expression": "contains()"` operation is similar however it's a logical-OR of all the groups that a user might be a part of which needs to be accounted for in the set membership. In this case we build up a `$expression` statement below for each group the user is in and then pass this to the filter:

```
#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)
    }
}
```

Writing data

Writing data on mutations is always controlled on the request mapping template. In the case of DynamoDB data sources, the key is to use an appropriate `"condition":{"expression"...}"` which performs validation against the authorization metadata in that table. In Security, we provided an example you can use to check the `Author` field in a table. The use cases in this section explore more use cases.

Use case: multiple owners

Using the example table diagram from earlier, suppose the `PeopleCanAccess` list

```
{
    "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)
        }
    }
}
```

Use case: group can create new record

```
#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)
    }
}
```

Use case: group can update existing record

```
#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)
    }
}
```

Public and private records

With the conditional filters you can also choose to mark data as private, public or some other Boolean check. This can then be combined as part of an authorization filter inside the response template. Using this check is a nice way to temporarily hide data or remove it from view without trying to control group membership.

For example suppose you added an attribute on each item in your DynamoDB table called `public` with either a value of `yes` or no. The following response template could be used on a `GetItem` call to only display data if the user is in a group that has access AND if that data is marked as public:

```
#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
```

The above code could also use a logical OR (`||`) to allow people to read if they have permission to a record or if it's public:

```
#if($hasPermission || $context.result.public == 'yes')
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

In general, you will find the standard operators `==`, `!=`, `&&`, and `||` helpful when performing authorization checks.

Real-time data

You can apply Fine Grained Access Controls to GraphQL subscriptions at the time a client makes a subscription, using the same techniques described earlier in this documentation. You attach a resolver to the subscription field, at which point you can query data from a data source and perform conditional logic in either the request or response mapping template. You can also return additional data to the client, such as the initial results from a subscription, as long as the data structure matches that of the returned type in your GraphQL subscription.

Use case: user can subscribe to specific conversations only

A common use case for real-time data with GraphQL subscriptions is building a messaging or private chat application. When creating a chat application that has multiple users, conversations can occur between two people or among multiple people. These might be grouped into “rooms”, which are private or public. As such, you would only want to authorize a user to subscribe to a conversation (which could be one to one or among a group) for which they have been granted access. For demonstration purposes, the sample below shows a simple use case of one user sending a private message to another. The setup has two Amazon DynamoDB tables:

- Messages table: (primary key) `toUser`, (sort key) `id`
- Permissions table: (primary key) `username`

The Messages table stores the actual messages that get sent via a GraphQL mutation. The Permissions table is checked by the GraphQL subscription for authorization at client connection time. The example below assumes you are using the following GraphQL schema:

```
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
}
```

Some of the standard operations, such as `createUserPermissions()`, are not covered below to illustrate the subscription resolvers, but are standard implementations of DynamoDB resolvers. Instead, we'll focus on subscription authorization flows with resolvers. To send a message from one user to another, attach a resolver to the `sendMessage()` field and select the **Messages** table data source with the following request template:

```
{
    "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),
    }
```

```
}
```

In this example, we use `$context.identity.username`. This returns user information for AWS Identity and Access Management or Amazon Cognito users. The response template is a simple passthrough of `$util.toJson($ctx.result)`. Save and go back to the schema page. Then attach a resolver for the `newMessage()` subscription, using the **Permissions** table as a data source and the following request mapping template:

```
{
    "version": "2018-05-29",
    "operation": "GetItem",
    "key": {
        "username": $util.dynamodb.toDynamoDBJson($ctx.identity.username),
    },
}
```

Then use the following response mapping template to perform your authorization checks using data from the **Permissions** table:

```
#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
```

In this case, you're doing three authorization checks. The first ensures that a result is returned. The second ensures that the user isn't subscribing to messages that are meant for another person. The third ensures that the user is allowed to subscribe to any fields, by checking a DynamoDB attribute of `isAuthorizedForSubscriptions` stored as a `BOOL`.

To test things out, you could sign in to the AWS AppSync console using Amazon Cognito user pools and a user named "Nadia", and then run the following GraphQL subscription:

```
subscription AuthorizedSubscription {
    newMessage(toUser: "Nadia") {
```

```
        id
        toUser
        fromUser
        content
    }
}
```

If in the **Permissions** table there is a record for the `username` key attribute of `Nadia` with `isAuthorizedForSubscriptions` set to `true`, you'll see a successful response. If you try a different `username` in the `newMessage()` query above, an error will be returned.

Using AWS WAF to protect your AWS AppSync APIs

AWS WAF is a web application firewall that helps protect web applications and APIs from attacks. It allows you to configure a set of rules, called a web access control list (web ACL), that allow, block, or monitor (count) web requests based on customizable web security rules and conditions that you define. When you integrate your AWS AppSync API with AWS WAF, you gain more control and visibility into the HTTP traffic accepted by your API. To learn more about AWS WAF, see How AWS WAF Works in the AWS WAF Developer Guide.

You can use AWS WAF to protect your AppSync API from common web exploits, such as SQL injection and cross-site scripting (XSS) attacks. These could affect API availability and performance, compromise security, or consume excessive resources. For example, you can create rules to allow or block requests from specified IP address ranges, requests from CIDR blocks, requests that originate from a specific country or region, requests that contain malicious SQL code, or requests that contain malicious script.

You can also create rules that match a specified string or a regular expression pattern in HTTP headers, method, query string, URI, and the request body (limited to the first 8 KB). Additionally, you can create rules to block attacks from specific user agents, bad bots, and content scrapers. For example, you can use rate-based rules to specify the number of web requests that are allowed by each client IP in a trailing, continuously updated, 5-minute period.

To learn more about the types of rules that are supported and additional AWS WAF features, see the AWS WAF Developer Guide and the AWS WAF API Reference.

> **Important**
>
> AWS WAF is your first line of defense against web exploits. When AWS WAF is enabled on an API, AWS WAF rules are evaluated before other access control features, such as API key authorization, IAM policies, OIDC tokens, and Amazon Cognito user pools.

Integrate an AppSync API with AWS WAF

You can integrate an Appsync API with AWS WAF using the AWS Management Console, the AWS CLI, AWS CloudFormation, or any other compatible client.

To integrate an AWS AppSync API with AWS WAF

1. Create an AWS WAF web ACL. For detailed steps using the AWS WAF Console, see Creating a web ACL.
2. Define the rules for the web ACL. A rule or rules are defined in the process of creating the web ACL. For information about how to structure rules, see AWS WAF rules. For examples of useful rules you can define for your AWS AppSync API, see Creating rules for a web ACL.
3. Associate the web ACL with an AWS AppSync API. You can perform this step in the AWS WAF Console or in the AppSync Console.
   - To associate the web ACL with an AWS AppSync API in the AWS WAF Console, follow the instructions for Associating or disassociating a Web ACL with an AWS resource in the AWS WAF Developer Guide.
   - To associate the web ACL with an AWS AppSync API in the AWS AppSync Console
     a. Sign in to the AWS Management Console and open the AppSync Console.
     b. Choose the API that you want to associate with a web ACL.
     c. In the navigation pane, choose **Settings**.
     d. In the **Web application firewall** section, turn on **Enable AWS WAF**.
     e. In the **Web ACL** dropdown list, choose the name of the web ACL to associate with your API.
     f. Choose **Save** to associate the web ACL with your API.

> **Note**
>
> After you create a web ACL in the AWS WAF Console, it can take a few minutes for the new web ACL to be available. If you do not see a newly created web ACL in the **Web application firewall** menu, wait a few minutes and retry the steps to associate the web ACL with your API.

> **Note**
>
> AWS WAF integration only supports the `Subscription registration message` event for real-time endpoints. AWS AppSync will respond with an error message instead of a `start_ack` message for any `Subscription registration message` blocked by AWS WAF.

After you associate a web ACL with an AWS AppSync API, you will manage the web ACL using the AWS WAF APIs. You do not need to re-associate the web ACL with your AWS AppSync API unless you want to associate the AWS AppSync API with a different web ACL.

Creating rules for a web ACL

Rules define how to inspect web requests and what to do when a web request matches the inspection criteria. Rules don't exist in AWS WAF on their own. You can access a rule by name in a rule group or in the web ACL where it's defined. For more information, see AWS WAF rules. The following examples demonstrate how to define and associate rules that are useful for protecting an AppSync API.

Example web ACL rule to limit request body size

The following is an example of a rule that limits the body size of requests. This would be entered into the **Rule JSON editor** when creating a web ACL in the AWS WAF Console.

```
{
    "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
        }
}
```

After you have created your web ACL using the preceding example rule, you must associate it with your AppSync API. As an alternative to using the AWS Management Console, you can perform this step in the AWS CLI by running the following command.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

It can take a few minutes for the changes to propagate, but after running this command, requests that contain a body larger than 1024 bytes will be rejected by AWS AppSync.

> **Note**
>
> After you create a new web ACL in the AWS WAF Console, it can take a few minutes for the web ACL to be available to associate with an API. If you run the CLI command and get a `WAFUnavailableEntityException` error, wait a few minutes and retry running the command.

Example web ACL rule to limit requests from a single IP address

The following is an example of a rule that throttles an AppSync API to 100 requests from a single IP address. This would be entered into the **Rule JSON editor** when creating a web ACL with a rate-based rule in the AWS WAF Console.

```
{
  "Name": "Throttle",
  "Priority": 0,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "Throttle"
  },
  "Statement": {
    "RateBasedStatement": {
      "Limit": 100,
      "AggregateKeyType": "IP"
    }
  }
}
```

After you have created your web ACL using the preceding example rule, you must associate it with your AppSync API. You can perform this step in the AWS CLI by running the following command.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

Example web ACL rule to prevent GraphQL __schema introspection queries to an API

The following is an example of a rule that prevents GraphQL __schema introspection queries to an API. Any HTTP body that includes the string "__schema" will be blocked. This would be entered into the **Rule JSON editor** when creating a web ACL in the AWS WAF Console.

```
{
  "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
        }
      ]
    }
  }
}
```

After you have created your web ACL using the preceding example rule, you must associate it with your AppSync API. You can perform this step in the AWS CLI by running the following command.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

Security in AWS AppSync

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to AWS AppSync, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using AWS AppSync. The following topics show you how to configure AWS AppSync to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your AWS AppSync resources.

Topics

Data protection in AWS AppSync

The AWS shared responsibility model applies to data protection in AWS AppSync. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with AWS AppSync or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Encryption in motion

AWS AppSync, like all AWS services, makes use of TLS1.2 and beyond for communication when using the AWS published APIs and SDKs.

Using AWS AppSync with other AWS services such as Amazon DynamoDB ensures encryption in transit: All AWS services use TLS 1.2 and beyond to communicate with one another unless otherwise specified. For resolvers that utilize Amazon EC2 or CloudFront, it is your responsibility to verify that TLS (HTTPS) is configured and secure. For information on configuring HTTPS in Amazon EC2, see Configuring SSL/TLS on Amazon Linux 2 in the Amazon EC2 user guide. For information about configuring HTTPS on CloudFront, see HTTPS in Amazon CloudFront in the CloudFront user guide.

Compliance validation for AWS AppSync

Third-party auditors assess the security and compliance of AWS AppSync as part of multiple AWS compliance programs. AWS AppSync is compliant with SOC, PCI, HIPAA/HIPAA BAA, IRAP, C5, ENS High, OSPAR, and HITRUST CSF programs.

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security Compliance & Governance – These solution implementation guides discuss architectural considerations and provide steps for deploying security and compliance features.
- HIPAA Eligible Services Reference – Lists HIPAA eligible services. Not all AWS services are HIPAA eligible.
- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Customer Compliance Guides – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map

the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).

- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see Security Hub controls reference.
- Amazon GuardDuty – This AWS service detects potential threats to your AWS accounts, workloads, containers, and data by monitoring your environment for suspicious and malicious activities. GuardDuty can help you address various compliance requirements, like PCI DSS, by meeting intrusion detection requirements mandated by certain compliance frameworks.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Infrastructure security in AWS AppSync

As a managed service, AWS AppSync is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well-Architected Framework*.

You use AWS published API calls to access AWS AppSync through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Resilience in AWS AppSync

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

In addition to the AWS global infrastructure, AWS AppSync allows most resources to be defined using AWS CloudFormation templates; for an example of using AWS CloudFormation templates to declare AWS AppSync resources, see Practical use cases for AWS AppSync Pipeline Resolvers on the AWS blog and the AWS CloudFormation User Guide.

Identity and access management for AWS AppSync

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use AWS AppSync resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- How AWS AppSync works with IAM
- Identity-based policies for AWS AppSync
- Troubleshooting AWS AppSync identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in AWS AppSync.

Service user – If you use the AWS AppSync service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more AWS AppSync features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in AWS AppSync, see Troubleshooting AWS AppSync identity and access.

Service administrator – If you're in charge of AWS AppSync resources at your company, you probably have full access to AWS AppSync. It's your job to determine which AWS AppSync features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with AWS AppSync, see How AWS AppSync works with IAM.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to AWS AppSync. To view example AWS AppSync identity-based policies that you can use in IAM, see Identity-based policies for AWS AppSync.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and AWS Multi-factor authentication in IAM in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. To temporarily assume an IAM role in the AWS Management Console, you can switch from a user to an IAM role (console). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Methods to assume a role in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Create a role for a third-party identity provider (federation) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or

store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.

- **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.
- **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.
- **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Use an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root user, regardless of whether they belong to your organization. For more information about Organizations and RCPs, including a list of AWS services that support RCPs, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's

permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How AWS AppSync works with IAM

Before you use IAM to manage access to AWS AppSync, learn what IAM features are available to use with AWS AppSync.

IAM features that you can use with AWS AppSync

IAM feature	AWS AppSync support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys	No
ACLs	No
ABAC (tags in policies)	Partial
Temporary credentials	Yes
Forward access sessions (FAS)	Partial
Service roles	No
Service-linked roles	Partial

To get a high-level view of how AWS AppSync and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for AWS AppSync

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for AWS AppSync

To view examples of AWS AppSync identity-based policies, see Identity-based policies for AWS AppSync.

Resource-based policies within AWS AppSync

Supports resource-based policies: No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by

attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Policy actions for AWS AppSync

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of AWS AppSync actions, see Actions defined by AWS AppSync in the *Service Authorization Reference*.

Policy actions in AWS AppSync use the following prefix before the action:

```
appsync
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "appsync:action1",
      "appsync:action2"
         ]
```

To view examples of AWS AppSync identity-based policies, see Identity-based policies for AWS AppSync.

Policy resources for AWS AppSync

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its Amazon Resource Name (ARN). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of AWS AppSync resource types and their ARNs, see Resources defined by AWS AppSync in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions defined by AWS AppSync.

To view examples of AWS AppSync identity-based policies, see Identity-based policies for AWS AppSync.

Policy condition keys for AWS AppSync

Supports service-specific policy condition keys: No

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM policy elements: variables and tags in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

To see a list of AWS AppSync condition keys, see Condition keys for AWS AppSync in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Actions defined by AWS AppSync.

To view examples of AWS AppSync identity-based policies, see Identity-based policies for AWS AppSync.

Access control lists (ACLs) in AWS AppSync

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Attribute-based access control (ABAC) with AWS AppSync

Supports ABAC (tags in policies): Partial

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see Define permissions with ABAC authorization in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using temporary credentials with AWS AppSync

Supports temporary credentials: Yes

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see AWS services that work with IAM in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see Switch from a user to an IAM role (console) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM.

Forward access sessions for AWS AppSync

Supports forward access sessions (FAS): Partial

When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

Service roles for AWS AppSync

Supports service roles: No

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

> **Warning**
>
> Changing the permissions for a service role might break AWS AppSync functionality. Edit service roles only when AWS AppSync provides guidance to do so.

Service-linked roles for AWS AppSync

Supports service-linked roles: Partial

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see AWS services that work with IAM in the *IAM User Guide*. Find a service in the table that includes a `Yes` in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policies for AWS AppSync

By default, users and roles don't have permission to create or modify AWS AppSync resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by AWS AppSync, including the format of the ARNs for each of the resource types, see Actions, resources, and condition keys for AWS AppSync in the *Service Authorization Reference*.

To learn the best practices for creating and configuring IAM identity-based policies, see the section called “IAM policy best practices”.

For a list of IAM identity-based policies for AWS AppSync, see AWS managed policies for AWS AppSync.

Topics

Using the AWS AppSync console

To access the AWS AppSync console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the AWS AppSync resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that IAM users and roles can still use the AWS AppSync console, also attach the AWS AppSync `ConsoleAccess` or `ReadOnly` AWS managed policy to the entities. For more information, see Adding permissions to a user in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "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": "*"
        }
    ]
}
```

Accessing one Amazon S3 bucket

In this example, you want to grant an IAM user in your AWS account access to one of your Amazon S3 buckets, `examplebucket`. You also want to allow the user to add, update, and delete objects.

In addition to granting the `s3:PutObject`, `s3:GetObject`, and `s3:DeleteObject` permissions to the user, the policy also grants the `s3:ListAllMyBuckets`, `s3:GetBucketLocation`, and `s3:ListBucket` permissions. These are the additional permissions required by the console. Also, the `s3:PutObjectAcl` and the `s3:GetObjectAcl` actions are required to be able to copy, cut, and paste objects in the console. For an example walkthrough that grants permissions to users and tests them using the console, see An example walkthrough: Using user policies to control access to your bucket.

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/*"
      }
   ]
}
```

Viewing AWS AppSync *widgets* based on tags

You can use conditions in your identity-based policy to control access to AWS AppSync resources based on tags. This example shows how you might create a policy that allows viewing a *widget*. However, permission is granted only if the *widget* tag `Owner` has the value of that user's user name. This policy also grants the permissions necessary to complete this action on the console.

You can attach this policy to the IAM users in your account. If a user named `richard-roe` attempts to view an AWS AppSync *widget*, the *widget* must be tagged `Owner=richard-roe` or `owner=richard-roe`. Otherwise he is denied access. The condition tag key `Owner` matches both

`Owner` and `owner` because condition key names are not case-sensitive. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.

AWS managed policies for AWS AppSync

To add permissions to users, groups, and roles, it is easier to use AWS managed policies than to write policies yourself. It takes time and expertise to create IAM customer managed policies that provide your team with only the permissions that they need. To get started quickly, you can use our AWS managed policies. These policies cover common use cases and are available in your AWS account. For more information about AWS managed policies, see AWS managed policies in the *IAM User Guide*.

AWS services maintain and update AWS managed policies. You can't change the permissions in AWS managed policies. Services occasionally add additional permissions to an AWS managed policy to support new features. This type of update affects all identities (users, groups, and roles) where the policy is attached. Services are most likely to update an AWS managed policy when a new feature is launched or when new operations become available. Services do not remove permissions from an AWS managed policy, so policy updates won't break your existing permissions.

Additionally, AWS supports managed policies for job functions that span multiple services. For example, the **ReadOnlyAccess** AWS managed policy provides read-only access to all AWS services and resources. When a service launches a new feature, AWS adds read-only permissions for new operations and resources. For a list and descriptions of job function policies, see AWS managed policies for job functions in the *IAM User Guide*.

AWS managed policy: AWSAppSyncInvokeFullAccess

Use the `AWSAppSyncInvokeFullAccess` AWS managed policy to allow your administrators to access the AWS AppSync service through the console or independently.

You can attach the `AWSAppSyncInvokeFullAccess` policy to your IAM identities.

Permissions details

This policy includes the following permissions.

- AWS AppSync – Allows full administrative access to all resources in AWS AppSync

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "appsync:GraphQL",
                "appsync:GetGraphqlApi",
                "appsync:ListGraphqlApis",
                "appsync:ListApiKeys"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policy: AWSAppSyncSchemaAuthor

Use the AWSAppSyncSchemaAuthor AWS managed policy to allow IAM users to access to create, update, and query their GraphQL schemas. For information about what users can do with these permissions, see Designing GraphQL APIs with AWS AppSync.

You can attach the AWSAppSyncSchemaAuthor policy to your IAM identities.

Permissions details

This policy includes the following permissions.

- AWS AppSync – Allows the following actions:
  - Creating GraphQL schemas
  - Allowing the creation, modification, and deletion of GraphQL types, resolvers, and functions

- Evaluating request and response template logic
- Evaluating code with a runtime and context
- Sending GraphQL queries to GraphQL APIs
- Retrieving GraphQL data

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 managed policy: AWSAppSyncPushToCloudWatchLogs

AWS AppSync uses Amazon CloudWatch to monitor the performance of your application by generating logs that you can use to troubleshoot and optimize your GraphQL requests. For more information, see Using CloudWatch to monitor and log GraphQL API data.

Use the `AWSAppSyncPushToCloudWatchLogs` AWS managed policy to allow AWS AppSync to push logs to an IAM user's CloudWatch account.

You can attach the `AWSAppSyncPushToCloudWatchLogs` policy to your IAM identities.

Permissions details

This policy includes the following permissions.

- `CloudWatch Logs` – Allows AWS AppSync to create log groups and streams with specified names. AWS AppSync pushes log events to the specified log stream.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
```

```
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policy: AWSAppSyncAdministrator

Use the `AWSAppSyncAdministrator` AWS managed policy to allow your administrators to access all of AWS AppSync except for the AWS console.

You can attach `AWSAppSyncAdministrator` to your IAM entities. AWS AppSync also attaches this policy to a service role that allows it to perform actions on your behalf.

Permissions details

This policy includes the following permissions.

- `AWS AppSync` – Allows full administrative access to all resources in AWS AppSync
- `IAM` – Allows the following actions:
  - Creating service-linked roles to allow AWS AppSync to analyze resources in other services on your behalf
  - Deleting service-linked roles
  - Passing service-linked roles on to other AWS services to assume the role later and to perform actions on your behalf

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 managed policy: AWSAppSyncServiceRolePolicy

Use the `AWSAppSyncServiceRolePolicy` AWS managed policy to allow access to AWS services and resources that AWS AppSync uses or manages.

You can't attach `AWSAppSyncServiceRolePolicy` to your IAM entities. This policy is attached to a service-linked role that allows AWS AppSync to perform actions on your behalf. For more information, see Service-linked roles for AWS AppSync.

Permissions details

This policy includes the following permissions.

- `X-Ray` – AWS AppSync uses AWS X-Ray to collect data about requests made within your application. For more information, see Using AWS X-Ray to trace requests in AWS AppSync.

  This policy allows the following actions:

  - Retrieving sampling rules and their results
  - Sending trace data to the X-Ray daemon

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:PutTraceSegments",
                "xray:PutTelemetryRecords",
                "xray:GetSamplingTargets",
                "xray:GetSamplingRules",
                "xray:GetSamplingStatisticSummaries"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
```

```
}
```

AWS AppSync updates to AWS managed policies

View details about updates to AWS managed policies for AWS AppSync since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the AWS AppSync Document history page.

Change	Description	Date
AWSAppSyncSchemaAuthor - Update to an existing policy	Added an `EvaluateCode` policy action to allow users to evaluate code with a runtime and context.	February 7, 2023
AWSAppSyncSchemaAuthor - Update to an existing policy	Added policy actions to allow the list, get, create, update, and delete functions for an API. Added an `EvaluateMappingTemplate` policy action to allow users to evaluate request and response resolver mapping template logic. Added policy actions to allow resource tagging.	August 25, 2022
AWS AppSync started tracking changes	AWS AppSync started tracking changes for its AWS managed policies.	August 25, 2022

Troubleshooting AWS AppSync identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with AWS AppSync and IAM.

I am not authorized to perform an action in AWS AppSync

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password.

The following example error occurs when the IAM user `mateojackson` tries to use the console to view details about a fictional *`my-example-widget`* resource, but he does not have the fictional `appsync:GetWidget` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 appsync:GetWidget on resource: my-example-widget
```

In this case, Mateo asks his administrator to update his policies to allow him to access the *`my-example-widget`* resource using the `appsync:GetWidget` action.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to AWS AppSync.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in AWS AppSync. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to view my access keys

After you create your IAM user access keys, you can view your access key ID at any time. However, you can't view your secret access key again. If you lose your secret key, you must create a new access key pair.

Access keys consist of two parts: an access key ID (for example, `AKIAIOSFODNN7EXAMPLE`) and a secret access key (for example, `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`). Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests. Manage your access keys as securely as you do your user name and password.

> **Important**
>
> Do not provide your access keys to a third party, even to help find your canonical user ID. By doing this, you might give someone permanent access to your AWS account.

When you create an access key pair, you are prompted to save the access key ID and secret access key in a secure location. The secret access key is available only at the time you create it. If you lose your secret access key, you must add new access keys to your IAM user. You can have a maximum of two access keys. If you already have two, you must delete one key pair before creating a new one. To view instructions, see Managing access keys in the *IAM User Guide*.

I'm an administrator and want to allow others to access AWS AppSync

To allow others to access AWS AppSync, you must grant permission to the people or applications that need access. If you are using AWS IAM Identity Center to manage people and applications, you assign permission sets to users or groups to define their level of access. Permission sets automatically create and assign IAM policies to IAM roles that are associated with the person or application. For more information, see Permission sets in the *AWS IAM Identity Center User Guide*.

If you are not using IAM Identity Center, you must create IAM entities (users or roles) for the people or applications that need access. You must then attach a policy to the entity that grants them the correct permissions in AWS AppSync. After the permissions are granted, provide the credentials to the user or application developer. They will use those credentials to access AWS. To learn more

about creating IAM users, groups, policies, and permissions, see IAM Identities and Policies and permissions in IAM in the *IAM User Guide*.

I want to allow people outside of my AWS account to access my AWS AppSync resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether AWS AppSync supports these features, see How AWS AppSync works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Logging AWS AppSync API calls with AWS CloudTrail

AWS AppSync is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in AWS AppSync. CloudTrail captures API calls for AWS AppSync as events. The calls captured include calls from the AWS AppSync console and code calls to the AWS AppSync API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for AWS AppSync. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to AWS AppSync, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

AWS AppSync information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in AWS AppSync, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing events with CloudTrail Event history.

For an ongoing record of events in your AWS account, including events for AWS AppSync, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple regions and Receiving CloudTrail log files from multiple accounts

AWS AppSync supports logging of calls made through the AWS AppSync API. At this time, calls to your APIs, as well as calls made to resolvers are not logged by AWS AppSync into CloudTrail.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity element.

Understanding AWS AppSync log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single

request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the `GetGraphqlApi` action made through the AWS AppSync console:


```
{
    "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"
}
```

The following example shows a CloudTrail log entry that demonstrates the `CreateApikey` action made through the AWS CLI:

```
{
    "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"
}
```

Security best practices for AWS AppSync

Securing AWS AppSync is more than simply turning on a few levers or setting up logging. The following sections discuss security best practices that vary depending on how you use the service.

Understand authentication methods

AWS AppSync provides multiple ways to authenticate your users to your GraphQL APIs. Each method has trade-offs in security, auditability, and usability.

The following common authentication methods are available:

- Amazon Cognito user pools allow your GraphQL API to use user attributes for fine-grained access control and filtering.
- API tokens have a limited lifetime and are appropriate for automated systems, such as Continuous Integration systems and integration with external APIs.
- AWS Identity and Access Management (IAM) is appropriate for internal applications managed in your AWS account.
- OpenID Connect allows you to control and federate access with the OpenID Connect protocol.

For more information on authentication and authorization in AWS AppSync, see Configuring authorization and authentication to secure your GraphQL APIs.

Understand how API configuration changes propagate

When you save changes to your API configuration, AWS AppSync starts to propagate the changes. Until your configuration change is propagated, AWS AppSync continues to serve your content from the previous configuration. After your configuration change is propagated, AWS AppSync immediately starts to serve your content based on the new configuration. While AWS AppSync is propagating your changes for an API, we can't determine whether the API is serving your content based on the previous configuration or the new configuration.

Use TLS for HTTP resolvers

When using HTTP resolvers, make sure to use TLS-secured (HTTPS) connections wherever possible. For a full list of the TLS certificates that AWS AppSync trusts, see Certificate Authorities (CA) Recognized by AWS AppSync for HTTPS Endpoints.

Use roles with the least permissions possible

When using resolvers such as the DynamoDB resolver, use roles that provide the most restrictive view to your resources, such as your Amazon DynamoDB tables.

IAM policy best practices

Identity-based policies determine whether someone can create, access, or delete AWS AppSync resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and

functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.

- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

AWS AppSync resolver reference (JavaScript)

The following sections contain the `APPSYNC_JS` runtime and JavaScript resolver reference:

- JavaScript resolvers overview - Learn more about how resolvers work in AWS AppSync.
- Resolver context object reference - Learn more about the context object and how it's used in resolvers.
- JavaScript runtime features for resolvers and functions - Learn more about supported runtime features and using utilities to simplify code.
- JavaScript resolver function reference for DynamoDB - Learn more about how resolvers interact with DynamoDB.
- JavaScript resolver function reference for OpenSearch - Learn more about resolver request and response structure and interactions with OpenSearch Service.
- JavaScript resolver function reference for Lambda - Learn more about resolver request and response structure and interactions with Lambda.
- JavaScript resolver function reference for EventBridge data source - Learn more about resolver request and response structure and interactions with EventBridge.
- JavaScript resolver function reference for None data source - Learn more about resolver request and response structure and interactions with NONE data sources.
- JavaScript resolver function reference for HTTP - Learn more about resolver request and response structure and interactions with HTTP endpoints.
- JavaScript resolver function reference for Amazon RDS - Learn more about resolver structure and interactions with RDS.
- JavaScript resolver function reference for Amazon Bedrock - Learn more about resolver structure and interactions with Amazon Bedrock.

AWS AppSync JavaScript resolvers overview

AWS AppSync lets you respond to GraphQL requests by performing operations on your data sources. For each GraphQL field you wish to run a query, mutation, or subscription on, a resolver must be attached.

Resolvers are the connectors between GraphQL and a data source. They tell AWS AppSync how to translate an incoming GraphQL request into instructions for your backend data source and how to

translate the response from that data source back into a GraphQL response. With AWS AppSync, you can write your resolvers using JavaScript and run them in the AWS AppSync (APPSYNC_JS) environment.

AWS AppSync allows you to write unit resolvers or pipeline resolvers composed of multiple AWS AppSync functions in a pipeline.

Supported runtime features

The AWS AppSync JavaScript runtime provides a subset of JavaScript libraries, utilities, and features. For a complete list of features and functionality supported by the APPSYNC_JS runtime, see JavaScript runtime features for resolvers and functions.

Unit resolvers

A unit resolver is composed of code that defines a request and response handler that are executed against a data source. The request handler takes a context object as an argument and returns the request payload used to call your data source. The response handler receives a payload back from the data source with the result of the executed request. The response handler transforms the payload into a GraphQL response to resolve the GraphQL field. In the example below, a resolver retrieves an item from an DynamoDB data source:

```
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;
```

Anatomy of a JavaScript pipeline resolver

A pipeline resolver is composed of code that defines a request and response handler and a list of functions. Each function has a **request** and **response** handler that it executes against a data source. As a pipeline resolver delegates runs to a list of functions, it is therefore not linked to any data source. Unit resolvers and functions are primitives that execute operation against data sources.

Pipeline resolver request handler

The request handler of a pipeline resolver (the before step) allows you to perform some preparation logic before running the defined functions.

Functions list

The list of functions a pipeline resolver will run in sequence. The pipeline resolver request handler evaluation result is made available to the first function as `ctx.prev.result`. Each function evaluation result is available to the next function as `ctx.prev.result`.

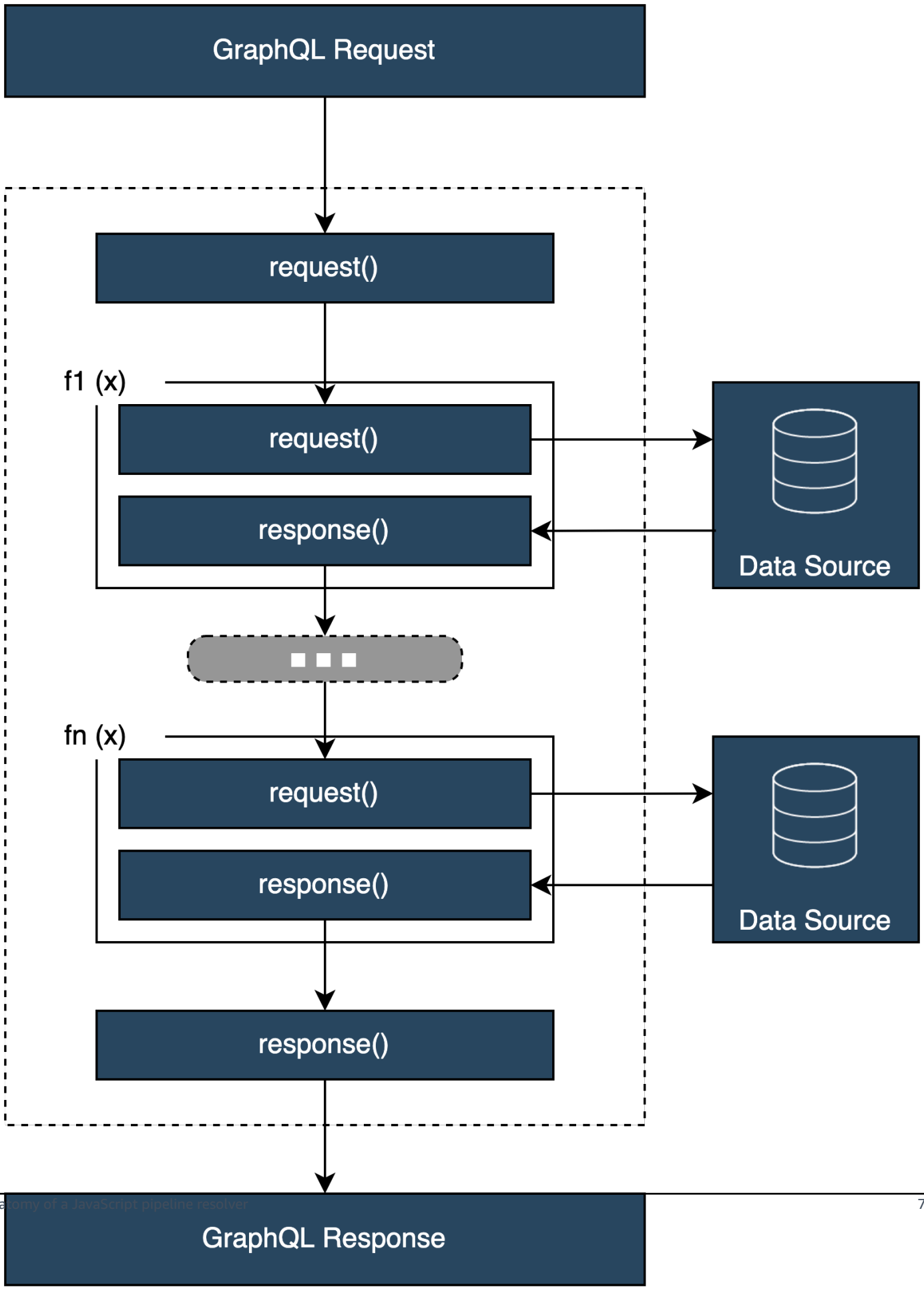
Pipeline resolver response handler

The response handler of a pipeline resolver allows you to perform some final logic from the output of the last function to the expected GraphQL field type. The output of the last function in the functions list is available in the pipeline resolver response handler as `ctx.prev.result` or `ctx.result`.

Execution flow

Given a pipeline resolver comprised of two functions, the list below represents the execution flow when the resolver is invoked:

1. Pipeline resolver request handler
2. Function 1: Function request handler
3. Function 1: Data source invocation
4. Function 1: Function response handler
5. Function 2: Function request handler
6. Function 2: Data source invocation
7. Function 2: Function response handler
8. Pipeline resolver response handler


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

Useful APPSYNC_JS runtime built-in utilities

The following utilities can help you when you're working with pipeline resolvers.

ctx.stash

The stash is an object that is made available inside each resolver and function request and response handler. The same stash instance lives through a single resolver run. This means that you can use the stash to pass arbitrary data across request and response handlers and across functions in a pipeline resolver. You can test the stash like a regular JavaScript object.

ctx.prev.result

The `ctx.prev.result` represents the result of the previous operation that was executed in the pipeline. If the previous operation was the pipeline resolver request handler, then `ctx.prev.result` is made available to the first function in the chain. If the previous operation was the first function, then `ctx.prev.result` represents the output of the first function and is made available to the second function in the pipeline. If the previous operation was the last function, then `ctx.prev.result` represents the output of the last function and is made available to the pipeline resolver response handler.

util.error

The `util.error` utility is useful to throw a field error. Using `util.error` inside a function request or response handler throws a field error immediately, which prevents subsequent functions from being executed. For more details and other `util.error` signatures, visit JavaScript runtime features for resolvers and functions.

util.appendError

`util.appendError` is similar to `util.error()`, with the major distinction that it doesn't interrupt the evaluation of the handler. Instead, it signals there was an error with the field, but allows the handler to be evaluated and consequently return data. Using `util.appendError` inside a function will not disrupt the execution flow of the pipeline. For more details and other `util.error` signatures, visit the JavaScript runtime features for resolvers and functions.

runtime.earlyReturn

The `runtime.earlyReturn` function allows you to prematurely return from any request function. Using `runtime.earlyReturn` inside of a resolver request handler will return from the resolver.

Calling it from an AWS AppSync function request handler will return from the function and will continue the run to either the next function in the pipeline or the resolver response handler.

Writing pipeline resolvers

A pipeline resolver also has a request and a response handler surrounding the run of the functions in the pipeline: its request handler is run before the first function's request, and its response handler is run after the last function's response. The resolver request handler can set up data to be used by functions in the pipeline. The resolver response handler is responsible for returning data that maps to the GraphQL field output type. In the example below, a resolver request handler, defines `allowedGroups`; the data returned should belong to one of these groups. This value can be used by the resolver's functions to request data. The resolver's response handler conducts a final check and filters the result to make sure that only items that belong to the allowed groups are returned.

```
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;
}
```

Writing AWS AppSync functions

AWS AppSync functions enable you to write common logic that you can reuse across multiple resolvers in your schema. For example, you can have one AWS AppSync function called `QUERY_ITEMS` that is responsible for querying items from an Amazon DynamoDB data source. For resolvers that you'd like to query items with, simply add the function to the resolver's pipeline and provide the query index to be used. The logic doesn't have to be re-implemented.

Supplemental topics

Topics

Example pipeline resolver with Amazon DynamoDB

Suppose you wanted to attach a pipeline resolver on a field named `getPost(id:ID!)` that returns a `Post` type from an Amazon DynamoDB data source with the following GraphQL query:

```
getPost(id:1){
    id
    title
    content
}
```

First, attach a simple resolver to `Query.getPost` with the code below. This is an example of simple resolver code. There is no logic defined in the request handler, and the response handler simply returns the result of the last function.

```
/**
 * 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
}
```

Next, define function `GET_ITEM` that retrieves a postitem from your data source:

```
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
}
```

If there is an error during the request, the function's response handler appends an error that will be returned to the calling client in the GraphQL response. Add the `GET_ITEM` function to your resolver functions list. When you execute the query, the `GET_ITEM` function's request handler uses the utils provided by AWS AppSync's DynamoDB module to create a `DynamoDBGetItem` request using the `id` as the key. `ddb.get({ key: { id } })` generates the appropriate `GetItem` operation:

```
{
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "1" }
    }
}
```

AWS AppSync uses the request to fetch the data from Amazon DynamoDB. Once the data is returned, it is handled by the `GET_ITEM` function's response handler, which checks for errors and then returns the result.

```
{
  "result" : {
    "id": 1,
    "title": "hello world",
    "content": "<long story>"
  }
}
```

Finally, the resolver's response handler returns the result directly.

Working with errors

If an error occurs in your function during a request, the error will be made available in your function response handler in `ctx.error`. You can append the error to your GraphQL response using the `util.appendError` utility. You can make the error available to other functions in the pipeline by using the stash. See the example below:

```
/**
```

```
 * 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;
}
```

Configuring utilities for the APPSYNC_JS runtime

AWS AppSync provides two libraries that aid in the development of resolvers with the APPSYNC_JS runtime:

- `@aws-appsync/eslint-plugin` - Catches and fixes problems quickly during development.
- `@aws-appsync/utils` - Provides type validation and autocompletion in code editors.

Configuring the eslint plugin

ESLint is a tool that statically analyzes your code to quickly find problems. You can run ESLint as part of your continuous integration pipeline. `@aws-appsync/eslint-plugin` is an ESLint plugin that catches invalid syntax in your code when leveraging the APPSYNC_JS runtime. The plugin allows you to quickly get feedback about your code during development without having to push your changes to the cloud.

`@aws-appsync/eslint-plugin` provides two rule sets that you can use during development.

"plugin:@aws-appsync/base" configures a base set of rules that you can leverage in your project:

Rule	Description
no-async	Async processes and promises are not supported.

Rule	Description
no-await	Async processes and promises are not supported.
no-classes	Classes are not supported.
no-for	`for` is not supported (except for `for-in` and `for-of`, which are supported)
no-continue	`continue` is not supported.
no-generators	Generators are not supported.
no-yield	`yield` is not supported.
no-labels	Labels are not supported.
no-this	`this` keyword is not supported.
no-try	Try/catch structure is not supported.
no-while	While loops are not supported.
no-disallowed-unary-operators	`++`, `--`, and ~ unary operators are not allowed.
no-disallowed-binary-operators	The `instanceof` operator is not allowed.
no-promise	Async processes and promises are not supported.

"plugin:@aws-appsync/recommended" provides some additional rules but also requires you to add TypeScript configurations to your project.

Rule	Description
no-recursion	Recursive function calls are not allowed.

Rule	Description
no-disallowed-methods	Some methods are not allowed. See the reference for a full set of supported built-in functions.
no-function-passing	Passing functions as function arguments to functions is not allowed.
no-function-reassign	Functions cannot be reassigned.
no-function-return	Functions cannot be the return value of functions.

To add the plugin to your project, follow the installation and usage steps at Getting Started with ESLint. Then, install the plugin in your project using your project package manager (e.g., npm, yarn, or pnpm):

```
$ npm install @aws-appsync/eslint-plugin
```

In your `.eslintrc.{js,yml,json}` file, add **"plugin:@aws-appsync/base"** or **"plugin:@aws-appsync/recommended"** to the `extends` property. The snippet below is a basic sample `.eslintrc` configuration for JavaScript:

```
{
  "extends": ["plugin:@aws-appsync/base"]
}
```

To use the **"plugin:@aws-appsync/recommended"** rule set, install the required dependency:

```
$ npm install -D @typescript-eslint/parser
```

Then, create an `.eslintrc.js` file:

```
{
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaVersion": 2018,
    "project": "./tsconfig.json"
```

```
  },
  "extends": ["plugin:@aws-appsync/recommended"]
}
```

Bundling, TypeScript, and source maps for the APPSYNC_JS runtime

TypeScript enhances AWS AppSync development by providing type safety and early error detection. You can write TypeScript code locally and transpile it to JavaScript before using it with the APPSYNC_JS runtime. The process starts with installing TypeScript and configuring tsconfig.json for the APPSYNC_JS environment. You can then use bundling tools like esbuild to compile and bundle the code. The Amplify CLI will generate types from the GraphQL schema, allowing you to use these types in resolver code.

You can leverage custom and external libraries in your resolver and function code, as long as they comply with APPSYNC_JS requirements. Bundling tools combine code into a single file for use in AWS AppSync. Source maps can be included to aid debugging.

Leveraging libraries and bundling your code

In your resolver and function code, you can leverage both custom and external libraries so long as they comply with the APPSYNC_JS requirements. This makes it possible to reuse existing code in your application. To make use of libraries that are defined by multiple files, you must use a bundling tool, such as esbuild, to combine your code in a single file that can then be saved to your AWS AppSync resolver or function.

When bundling your code, keep the following in mind:

- APPSYNC_JS only supports ECMAScript modules (ESM).
- `@aws-appsync/*` modules are integrated into APPSYNC_JS and should not be bundled with your code.
- The APPSYNC_JS runtime environment is similar to NodeJS in that code does not run in a browser environment.
- You can include an optional source map. However, do not include the source content.

  To learn more about source maps, see Using source maps.

For example, to bundle your resolver code located at `src/appsync/getPost.resolver.js`, you can use the following esbuild CLI command:

```
$ 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
```

Building your code and working with TypeScript

TypeScript is a programming language developed by Microsoft that offers all of JavaScript's features along with the TypeScript typing system. You can use TypeScript to write type-safe code and catch errors and bugs at build time before saving your code to AWS AppSync. The `@aws-appsync/utils` package is fully typed.

The APPSYNC_JS runtime doesn't support TypeScript directly. You must first transpile your TypeScript code to JavaScript code that the APPSYNC_JS runtime supports before saving your code to AWS AppSync. You can use TypeScript to write your code in your local integrated development environment (IDE), but note that you cannot create TypeScript code in the AWS AppSync console.

To get started, make sure you have TypeScript installed in your project. Then, configure your TypeScript transcompilation settings to work with the APPSYNC_JS runtime using TSConfig. Here's an example of a basic `tsconfig.json` file that you can use:

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "esnext",
    "module": "esnext",
   "noEmit": true,
   "moduleResolution": "node",
  }
}
```

You can then use a bundling tool like esbuild to compile and bundle your code. For example, given a project with your AWS AppSync code located at `src/appsync`, you can use the following command to compile and bundle your code:

```
$ esbuild --bundle \
--sourcemap=inline \
--sources-content=false \
--target=esnext \
--platform=node \
--format=esm \
--external:@aws-appsync/utils \
--outdir=out/appsync \
 src/appsync/**/*.ts
```

Using Amplify codegen

You can use the Amplify CLI to generate the types for your schema. From the directory where your `schema.graphql` file is located, run the following command and review the prompts to configure your codegen:

```
$  npx @aws-amplify/cli codegen add
```

To regenerate your codegen under certain circumstances (e.g., when your schema is updated), run the following command:

```
$ npx @aws-amplify/cli codegen
```

You can then use the generated types in your resolver code. For example, given the following schema:

```
type Todo {
  id: ID!
  title: String!
  description: String
}

type Mutation {
  createTodo(title: String!, description: String): Todo
}

type Query {
  listTodos: Todo
}
```

You could use the generated types in the following example AWS AppSync function:

```
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
}
```

Using generics in TypeScript

You can use generics with several of the provided types. For example, the snippet below is a Todo type:

```
export type Todo = {
  __typename: "Todo",
  id: string,
  title: string,
  description?: string | null,
};
```

You can write a resolver for a subscription that makes use of Todo. In your IDE, type definitions and auto-complete hints will guide you into properly using the `toSubscriptionFilter` transform utility:

```
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
}
```

Linting your bundles

You can automatically lint your bundles by importing the `esbuild-plugin-eslint` plugin. You can then enable it by providing a `plugins` value that enables eslint capabilities. Below is a snippet that uses the esbuild JavaScript API in a file called `build.mjs`:

```
/* 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 })],
})
```

Using source maps

You can provide an inline source map (`sourcemap`) with your JavaScript code. Source maps are useful for when you bundle JavaScript or TypeScript code and want to see references to your input source files in your logs and runtime JavaScript error messages.

Your `sourcemap` must appear at the end of your code. It is defined by a single comment line that follows the following format:

```
//# sourceMappingURL=data:application/json;base64,<base64 encoded string>
```

Here's an example:

```
//# sourceMappingURL=data:application/
json;base64,ewogICJ2ZXJzaW9uIjogMywKICAic291cmNlcyI6IFsibGliLmpzIiwgImNvZGUuanMiXSwKICAibWFwcGl
```

Source maps can be created with esbuild. The example below shows you how to use the esbuild JavaScript API to include an inline source map when code is built and bundled:

```
/* 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 })],
})
```

In particular, the `sourcemap` and `sourcesContent` options specify that a source map should be added in line at the end of each build but should not include the source content. As a convention, we recommend not including source content in your `sourcemap`. You can disable this in esbuild by setting `sources-content` to `false`.

To illustrate how source maps work, review the following example in which a resolver code references helper functions from a helper library. The code contains log statements in the resolver code and in the helper library:

./src/default.resolver.ts (your resolver)

```
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 (a helper file)

```
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..')
}
```

When you build and bundle the resolver file, your resolver code will include an inline source map. When your resolver runs, the following entries appear in the CloudWatch logs:

Looking at the entries in the CloudWatch log, you'll notice that the functionality of the two files have been bundled together and are running concurrently. The original file name of each file is also clearly reflected in the logs.

Testing your resolver and function handlers in AWS AppSync

You can use the `EvaluateCode` API command to remotely test your resolver and function handlers with mocked data before ever saving your code to a resolver or function. To get started with the command, make sure you have added the `appsync:evaluatecode` permission to your policy. For example:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateCode",
            "Resource": "arn:aws:appsync:us-east-1:111122223333:*"
        }
    ]
}
```

You can leverage the command by using the AWS CLI or AWS SDKs. For example, to test your code using the CLI, simply point to your file, provide a context, and specify the handler you want to evaluate:

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

The response contains an `evaluationResult` containing the payload returned by your handler. It also contains a `logs` object that holds the list of logs that were generated by your handler during the evaluation. This makes it easy to debug your code execution and see information about your evaluation to help troubleshoot. For example:

```
{
    "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\""
    ]
}
```

The evaluation result can be parsed as JSON, which gives:

```
{
  "operation": "PutItem",
  "key": {
    "id": {
      "S": "record-id"
    }
  },
  "attributeValues": {
    "owner": {
      "S": "John doe"
    },
    "expectedVersion": {
      "N": 2
    },
    "authorId": {
      "S": "Sammy Davis"
    }
  }
}
```

Using the SDK, you can easily incorporate tests from your test suite to validate your code's behavior. Our example here uses the Jest Testing Framework, but any testing suite works. The following snippet shows a hypothetical validation run. Note that we expect the evaluation response to be valid JSON, so we use `JSON.parse` to retrieve JSON from the string response:

```
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)
})
```

This yields the following result:

```
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
```

Migrating from VTL to JavaScript in AWS AppSync

AWS AppSync allows you to write your business logic for your resolvers and functions using VTL or JavaScript. With both languages, you write logic that instructs the AWS AppSync service on how to interact with your data sources. With VTL, you write mapping templates that must evaluate to a valid JSON-encoded string. With JavaScript, you write request and response handlers that return objects. You don't return a JSON-encoded string.

For example, take the following VTL mapping template to get an Amazon DynamoDB item:

```
{
    "operation": "GetItem",
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
    }
}
```

The utility `$util.dynamodb.toDynamoDBJson` returns a JSON-encoded string. If `$ctx.args.id` is set to `<id>`, the template evaluates to a valid JSON-encoded string:

```
{
    "operation": "GetItem",
    "key": {
        "id": {"S": "<id>"},
    }
}
```

When working with JavaScript, you do not need to print out raw JSON-encoded strings within your code, and using a utility like `toDynamoDBJson` is not needed. An equivalent example of the mapping template above is:

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: {id: util.dynamodb.toDynamoDB(ctx.args.id)}
  };
}
```

An alternative is to use `util.dynamodb.toMapValues`, which is the recommended approach to handle an object of values:

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}
```

This evaluates to:

```
{
  "operation": "GetItem",
  "key": {
    "id": {
      "S": "<id>"
    }
  }
}
```

Note

We recommend using the DynamoDB module with DynamoDB data sources:

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 ddb.get({ key: { id: ctx.args.id } })
}
```

As another example, take the following mapping template to put an item in an Amazon DynamoDB data source:

```
{
    "operation" : "PutItem",
    "key" : {
        "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

When evaluated, this mapping template string must produce a valid JSON-encoded string. When using JavaScript, your code returns the request object directly:

```
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),
  };
}
```

which evaluates to:

```
{
  "operation": "PutItem",
  "key": {
    "id": { "S": "2bff3f05-ff8c-4ed8-92b4-767e29fc4e63" }
  },
  "attributeValues": {
    "firstname": { "S": "Shaggy" },
    "age": { "N": 4 }
  }
}
```

Note

We recommend using the DynamoDB module with DynamoDB data sources:

```
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 })
}
```

Choosing between direct data source access and proxying via a Lambda data source

With AWS AppSync and the APPSYNC_JS runtime, you can write your own code that implements your custom business logic by using AWS AppSync functions to access your data sources. This makes it easy for you to directly interact with data sources like Amazon DynamoDB, Aurora Serverless, OpenSearch Service, HTTP APIs, and other AWS services without having to deploy additional computational services or infrastructure. AWS AppSync also makes it easy to interact with an AWS Lambda function by configuring a Lambda data source. Lambda data sources allow you to run complex business logic using AWS Lambda's full set capabilities to resolve a GraphQL request. In most cases, an AWS AppSync function directly connected to its target data source will provide all of the functionality you need. In situations where you need to implement complex business logic that is not supported by the APPSYNC_JS runtime, you can use a Lambda data source as a proxy to interact with your target data source.

	Direct data source integration	**Lambda data source as a proxy**
Use case	AWS AppSync functions interact directly with API data sources.	AWS AppSync functions call Lambdas that interact with API data sources.
Runtime	*APPSYNC_JS (JavaScript)*	*Any supported Lambda runtime*
Maximum size of code	*32,000 characters per AWS AppSync function*	*50 MB (zipped, for direct upload) per Lambda*

External modules	*Limited - APPSYNC_JS supported features only*	*Yes*
Call any AWS service	*Yes - Using AWS AppSync HTTP datasource*	*Yes - Using AWS SDK*
Access to the request header	*Yes*	*Yes*
Network access	*No*	*Yes*
File system access	*No*	*Yes*
Logging and metrics	*Yes*	*Yes*
Build and test entirely within AppSync	*Yes*	*No*
Cold start	*No*	*No - With provisioned concurrency*
Auto-scaling	*Yes - transparently by AWS AppSync*	*Yes - As configured in Lambda*
Pricing	*No additional charge*	*Charged for Lambda usage*

AWS AppSync functions that integrate directly with their target data source are ideal for use cases like the following:

- Interacting with Amazon DynamoDB, Aurora Serverless, and OpenSearch Service
- Interacting with HTTP APIs and passing incoming headers
- Interacting with AWS services using HTTP data sources (with AWS AppSync automatically signing requests with the provided data source role)
- Implementing access control before accessing data sources
- Implementing the filtering of retrieved data prior to fulfilling a request
- Implementing simple orchestration with sequential execution of AWS AppSync functions in a resolver pipeline
- Controlling caching and subscription connections in queries and mutations.

AWS AppSync functions that use a Lambda data source as a proxy are ideal for use cases like the following:

- Using a language other than JavaScript or Velocity Template Language (VTL)
- Adjusting and controlling CPU or memory to optimize performance
- Importing third-party libraries or requiring unsupported features in `APPSYNC_JS`
- Making multiple network requests and/or getting file system access to fulfill a query
- Batching requests using batching configuration.

AWS AppSync JavaScript resolver context object reference

AWS AppSync defines a set of variables and functions for working with request and response handlers. This makes logical operations on data easier with GraphQL. This document describes those functions and provides examples.

Accessing the `context`

The `context` argument of a request and response handler is an object that holds all of the contextual information for your resolver invocation. It has the following structure:

```
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**
>
> You will often find that the `context` object is referred to as `ctx`.

Each field in the `context` object is defined as follows:

`context` fields

`arguments`

A map that contains all GraphQL arguments for this field.

`identity`

An object that contains information about the caller. For more information about the structure of this field, see Identity.

`source`

A map that contains the resolution of the parent field.

`stash`

The stash is an object that is made available inside each resolver and function handler. The same stash object lives through a single resolver run. This means that you can use the stash to pass arbitrary data across request and response handlers and across functions in a pipeline resolver.

> **Note**
>
> You cannot delete or replace the entire stash, but you can add, update, delete, and read properties of the stash.

You can add items to the stash by modifying one of the code examples below:

```
//Example 1
ctx.stash.newItem = { key: "something" }

//Example 2
Object.assign(ctx.stash, {key1: value1, key2: value})
```

You can remove items from the stash by modifying the code below:

```
delete ctx.stash.key
```

`result`

A container for the results of this resolver. This field is available only to response handlers.

For example, if you're resolving the `author` field of the following query:

```
query {
    getPost(id: 1234) {
        postId
        title
        content
        author {
            id
            name
        }
    }
}
```

Then the full `context` variable is available when a response handler is evaluated:

```
{
  "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`

The result of whatever previous operation was executed in a pipeline resolver.

If the previous operation was the pipeline resolver's request handler, then `ctx.prev.result` represents that evaluation result and is made available to the first function in the pipeline.

If the previous operation was the first function, then `ctx.prev.result` represents the evaluation result of the first function response handler and is made available to the second function in the pipeline.

If the previous operation was the last function, then `ctx.prev.result` represents the evaluation result of the last function and is made available to the pipeline resolver's response handler.

`info`

An object that contains information about the GraphQL request. For the structure of this field, see Info.

Identity

The `identity` section contains information about the caller. The shape of this section depends on the authorization type of your AWS AppSync API.

For more information about AWS AppSync security options, see Authorization and authentication.

`API_KEY` authorization

The `identity` field isn't populated.

`AWS_LAMBDA` authorization

The `identity` has the following form:

```
type AppSyncIdentityLambda = {
  resolverContext: any;
```

```
};
```

The `identity` contains the `resolverContext` key, containing the same `resolverContext` content returned by the Lambda function authorizing the request.

`AWS_IAM` authorization

The `identity` has the following form:

```
type AppSyncIdentityIAM = {
  accountId: string;
  cognitoIdentityPoolId: string;
  cognitoIdentityId: string;
  sourceIp: string[];
  username: string;
  userArn: string;
  cognitoIdentityAuthType: string;
  cognitoIdentityAuthProvider: string;
};
```

`AMAZON_COGNITO_USER_POOLS` authorization

The `identity` has the following form:

```
type AppSyncIdentityCognito = {
  sourceIp: string[];
  username: string;
  groups: string[] | null;
  sub: string;
  issuer: string;
  claims: any;
  defaultAuthStrategy: string;
};
```

Each field is defined as follows:

`accountId`

The AWS account ID of the caller.

`claims`

The claims that the user has.

`cognitoIdentityAuthType`

Either authenticated or unauthenticated based on the identity type.

`cognitoIdentityAuthProvider`

A comma-separated list of external identity provider information used in obtaining the credentials used to sign the request.

`cognitoIdentityId`

The Amazon Cognito identity ID of the caller.

`cognitoIdentityPoolId`

The Amazon Cognito identity pool ID associated with the caller.

`defaultAuthStrategy`

The default authorization strategy for this caller (`ALLOW` or `DENY`).

`issuer`

The token issuer.

`sourceIp`

The source IP address of the caller that AWS AppSync receives. If the request doesn't include the `x-forwarded-for` header, the source IP value contains only a single IP address from the TCP connection. If the request includes a `x-forwarded-for` header, the source IP is a list of IP addresses from the `x-forwarded-for` header, in addition to the IP address from the TCP connection.

`sub`

The UUID of the authenticated user.

`user`

The IAM user.

`userArn`

The Amazon Resource Name (ARN) of the IAM user.

`username`

The user name of the authenticated user. In the case of `AMAZON_COGNITO_USER_POOLS` authorization, the value of *username* is the value of attribute *cognito:username*. In the case of

`AWS_IAM` authorization, the value of *username* is the value of the AWS user principal. If you're using IAM authorization with credentials vended from Amazon Cognito identity pools, we recommend that you use `cognitoIdentityId`.

Access request headers

AWS AppSync supports passing custom headers from clients and accessing them in your GraphQL resolvers by using `ctx.request.headers`. You can then use the header values for actions such as inserting data into a data source or authorization checks. You can use single or multiple request headers using `$curl` with an API key from the command line, as shown in the following examples:

Single header example

Suppose you set a header of `custom` with a value of `nadia` like the following:

```
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
```

This could then be accessed with `ctx.request.headers.custom`. For example, it might be in the following code for DynamoDB:

```
"custom": util.dynamodb.toDynamoDB(ctx.request.headers.custom)
```

Multiple header example

You can also pass multiple headers in a single request and access these in the resolver handler. For example, if the `custom` header is set with two values:

```
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
```

You could then access these as an array, such as `ctx.request.headers.custom[1]`.

> **Note**
>
> AWS AppSync doesn't expose the cookie header in `ctx.request.headers`.

Access the request custom domain name

AWS AppSync supports configuring a custom domain that you can use to access your GraphQL and real-time endpoints for your APIs. When making a request with a custom domain name, you can get the domain name using `ctx.request.domainName`.

When using the default GraphQL endpoint domain name, the value is `null`.

Info

The `info` section contains information about the GraphQL request. This section has the following form:

```
type Info = {
  fieldName: string;
  parentTypeName: string;
  variables: any;
  selectionSetList: string[];
  selectionSetGraphQL: string;
};
```

Each field is defined as follows:

`fieldName`

The name of the field that is currently being resolved.

`parentTypeName`

The name of the parent type for the field that is currently being resolved.

`variables`

A map which holds all variables that are passed into the GraphQL request.

`selectionSetList`

A list representation of the fields in the GraphQL selection set. Fields that are aliased are referenced only by the alias name, not the field name. The following example shows this in detail.

`selectionSetGraphQL`

A string representation of the selection set, formatted as GraphQL schema definition language (SDL). Although fragments aren't merged into the selection set, inline fragments are preserved, as shown in the following example.

> **Note**
>
> `JSON.stringify` will not include `selectionSetGraphQL` and `selectionSetList` in the string serialization. You must reference these properties directly.

For example, if you are resolving the `getPost` field of the following query:

```
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
  }
}
```

Then the full `ctx.info` variable that is available when processing a handler might be:

```
{
  "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` exposes only fields that belong to the current type. If the current type is an interface or union, only selected fields that belong to the interface are exposed. For example, given the following schema:

```
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
}
```

And the following query:

```
query {
    node(id: "post1") {
        id
        ... on Post {
            title
        }

        ... on Blog {
            title
        }
    }
}
```

When calling `ctx.info.selectionSetList` at the `Query.node` field resolution, only `id` is exposed:

```
"selectionSetList": [
    "id"
]
```

AWS AppSync JavaScript runtime features for resolvers and functions

The APPSYNC_JS runtime environment provides functionality similar to ECMAScript (ES) version 6.0. It supports a subset of its features and provides some additional methods (utilities) that are not part of the ES specifications. The following topics list all the supported language features:

- Supported runtime features - Learn more about supported core features, primitive objects, built-in objects and functions, etc.
- Built-in utilities - The util variable contains general utility methods to help you work with data. Unless otherwise specified, all utilities use the UTF-8 character set.
- Built-in modules - Learn more about how built-in modules can help write JavaScript resolvers and functions.
- Runtime utilities - The runtime library provides utilities to control or modify the runtime properties of your resolvers and functions.
- Time helpers in util.time - The util.time variable contains datetime methods to help generate timestamps, convert between datetime formats, and parse datetime strings. The syntax for datetime formats is based on DateTimeFormatter, which you can reference for further documentation.
- DynamoDB helpers in util.dynamodb - util.dynamodb contains helper methods that make it easier to write and read data to Amazon DynamoDB, such as automatic type mapping and formatting.
- HTTP helpers in util.http - The util.http utility provides helper methods that you can use to manage HTTP request parameters and to add response headers.
- Transformation helpers in util.transform - util.transform contains helper methods that make it easier to perform complex operations against data sources.
- String helpers in util.str - util.str contains methods to help with common String operations.
- Extensions - extensions contains a set of methods to make additional actions within your resolvers.
- XML helpers in util.xml - util.xml contains methods to help with XML string conversion.

> **Note**
>
> Currently, this reference only applies to runtime version **1.0.0**.

Supported runtime features

The sections below describe the supported feature set of the APPSYNC_JS runtime.

Core features

The following core features are supported.

Types

The following types are supported:

- numbers
- strings
- booleans
- objects
- arrays
- functions

Operators

Operators are supported, including:

- standard math operators (`+`, `-`, `/`, `%`, `*`, etc.)
- nullish coalescing operator (`??`)
- Optional chaining (`?.`)
- bitwise operators
- `void` and `typeof` operators
- spread operators (`...`)

The following operators are not supported:

- unary operators (`++`, `--`, and `~`)
- `in` operator

> **Note**
>
> Use the `Object.hasOwn` operator to check if the specified property is in the specified object.

Statements

The following statements are supported:

- `const`
- `let`
- `var`
- `break`
- `else`
- `for-in`
- `for-of`
- `if`
- `return`
- `switch`
- spread syntax

The following are not supported:

- `catch`
- `continue`
- `do-while`
- `finally`
- `for(initialization; condition; afterthought)`

> **Note**
>
> The exceptions are `for-in` and `for-of` expressions, which are supported.

- `throw`
- `try`
- `while`
- labeled statements

Literals

The following ES 6 template literals are supported:

- Multi-line strings
- Expression interpolation
- Nesting templates

Functions

The following function syntax is supported:

- Function declarations are supported.
- ES 6 arrow functions are supported.
- ES 6 rest parameter syntax is supported.

Strict mode

Functions operate in strict mode by default, so you don't need to add a `use_strict` statement in your function code. This cannot be changed.

Primitive objects

The following primitive objects of ES and their functions are supported.

Object

The following objects are supported:

- `Object.assign()`
- `Object.entries()`
- `Object.hasOwn()`
- `Object.keys()`
- `Object.values()`
- `delete`

String

The following strings are supported:

- `String.prototype.length()`
- `String.prototype.charAt()`
- `String.prototype.concat()`
- `String.prototype.endsWith()`
- `String.prototype.indexOf()`
- `String.prototype.lastIndexOf()`
- `String.raw()`
- `String.prototype.replace()`

> **Note**
>
> Regular expressions are not supported.
> However, Java-styled regular expression constructs are supported in the provided parameter. For more information see Pattern.

- `String.prototype.replaceAll()`

> **Note**
>
> Regular expressions are not supported.
> However, Java-styled regular expression constructs are supported in the provided parameter. For more information see Pattern.

- `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

The following numbers are supported:

- `Number.isFinite`
- `Number.isNaN`

Built-in objects and functions

The following functions and objects are supported.

Math

The following math functions are supported:

- `Math.random()`
- `Math.min()`
- `Math.max()`
- `Math.round()`
- `Math.floor()`
- `Math.ceil()`

Array

The following array methods are supported:

- `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()` doesn't support arguments.

- `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

The console object is available for debugging. During live query execution, console log/error statements are sent to Amazon CloudWatch Logs (if logging is enabled). During code evaluation with `evaluateCode`, log statements are returned in the command response.

- `console.error()`
- `console.log()`

Function

- The `apply`, `bind`, and `call` methods not are supported.
- Function constructors are not supported.
- Passing a function as an argument is not supported.
- Recursive function calls are not supported.

JSON

The following JSON methods are supported:

- `JSON.parse()`

  > **Note**
  >
  > Returns a blank string if the parsed string is not valid JSON.

- `JSON.stringify()`

Promises

Async processes are not supported, and promises are not supported.

> **Note**
>
> Network and file system access is not supported within the `APPSYNC_JS` runtime in AWS AppSync. AWS AppSync handles all I/O operations based on the requests made by the AWS AppSync resolver or AWS AppSync function.

Globals

The following global constants are supported:

- NaN

- `Infinity`
- `undefined`
- `util`
- `extensions`
- `runtime`

Error types

Throwing errors with `throw` is not supported. You can return an error by using `util.error()` function. You can include an error in your GraphQL response by using the `util.appendError` function.

For more information, see Error utils.

Built-in utilities

The `util` variable contains general utility methods to help you work with data. Unless otherwise specified, all utilities use the UTF-8 character set.

Encoding utils

Encoding utils list

`util.urlEncode(String)`

Returns the input string as an `application/x-www-form-urlencoded` encoded string.

`util.urlDecode(String)`

Decodes an `application/x-www-form-urlencoded` encoded string back to its non-encoded form.

`util.base64Encode(string) : string`

Encodes the input into a base64-encoded string.

`util.base64Decode(string) : string`

Decodes the data from a base64-encoded string.

ID generation utils

ID generation utils list

`util.autoId()`

Returns a 128-bit randomly generated UUID.

`util.autoUlid()`

Returns a 128-bit randomly generated ULID (Universally Unique Lexicographically Sortable Identifier).

`util.autoKsuid()`

Returns a 128-bit randomly generated KSUID (K-Sortable Unique Identifier) base62 encoded as a String with a length of 27.

Error utils

Error utils list

`util.error(String, String?, Object?, Object?)`

Throws a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally, an `errorType` field, a `data` field, and an `errorInfo` field can be specified. The `data` value will be added to the corresponding `error` block inside `errors` in the GraphQL response.

> **Note**
>
> `data` will be filtered based on the query selection set. The `errorInfo` value will be added to the corresponding `error` block inside `errors` in the GraphQL response. `errorInfo` will **not** be filtered based on the query selection set.

`util.appendError(String, String?, Object?, Object?)`

Appends a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally,

an `errorType` field, a `data` field, and an `errorInfo` field can be specified. Unlike `util.error(String, String?, Object?, Object?)`, the template evaluation will not be interrupted, so that data can be returned to the caller. The `data` value will be added to the corresponding `error` block inside `errors` in the GraphQL response.

> **Note**
>
> `data` will be filtered based on the query selection set. The `errorInfo` value will be added to the corresponding `error` block inside `errors` in the GraphQL response. `errorInfo` will **not** be filtered based on the query selection set.

Type and pattern matching utils

Type and pattern matching utils list

`util.matches(String, String) : Boolean`

Returns true if the specified pattern in the first argument matches the supplied data in the second argument. The pattern must be a regular expression such as `util.matches("a*b", "aaaaab")`. The functionality is based on Pattern, which you can reference for further documentation.

`util.authType()`

Returns a String describing the multi-auth type being used by a request, returning back either "IAM Authorization", "User Pool Authorization", "Open ID Connect Authorization", or "API Key Authorization".

Return value behavior utils

Return value behavior utils list

`util.escapeJavaScript(String)`

Returns the input string as a JavaScript escaped string.

Resolver authorization utils

Resolver authorization utils list

`util.unauthorized()`

Throws `Unauthorized` for the field being resolved. Use this in request or response mapping templates to determine whether to allow the caller to resolve the field.

Built-in modules

Modules are a part of the `APPSYNC_JS` runtime and provide utilities to help write JavaScript resolvers and functions. For samples and examples, see the aws-appsync-resolver-samples GitHub repository.

DynamoDB module functions

DynamoDB module functions provide an enhanced experience when interacting with DynamoDB data sources. You can make requests toward your DynamoDB data sources using the functions and without adding type mapping.

Modules are imported using `@aws-appsync/utils/dynamodb`:

```
// Modules are imported using @aws-appsync/utils/dynamodb
import * as ddb from '@aws-appsync/utils/dynamodb';
```

Functions

Functions list

`get<T>(payload: GetInput): DynamoDBGetItemRequest`

> **Tip**
>
> See the section called “Inputs” for information about GetInput.

Generates a `DynamoDBGetItemRequest` object to make a GetItem request to DynamoDB.

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

```
export function request(ctx) {
 return get({ key: { id: ctx.args.id } });
}
```

put<T>(payload): DynamoDBPutItemRequest

Generates a `DynamoDBPutItemRequest` object to make a PutItem request to DynamoDB.

```
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

Generates a `DynamoDBDeleteItemRequest` object to make a DeleteItem request to DynamoDB.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 return ddb.remove({ key: { id: ctx.args.id } });
}
```

scan<T>(payload): DynamoDBScanRequest

Generates a `DynamoDBScanRequest` to make a Scan request to DynamoDB.

```
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

Generates a `DynamoDBSyncRequest` object to make a Sync request. The request only receives the data altered since the last query (delta updates). Requests can only be made to versioned DynamoDB data sources.

```
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`

Generates a `DynamoDBUpdateItemRequest` object to make an UpdateItem request to DynamoDB.

Operations

Operation helpers allow you to take specific actions on parts of your data during updates. To get started, import `operations` from `@aws-appsync/utils/dynamodb`:

```
// Modules are imported using operations
import {operations} from '@aws-appsync/utils/dynamodb';
```

Operations list

`add<T>(payload)`

A helper function that adds a new attribute item when updating DynamoDB.

Example

To add an address (street, city, and zip code) to an existing DynamoDB item using the ID value:

```
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)`

A helper function that appends a payload to the existing list in DynamoDB.

Example

To append newly added friend IDs (`newFriendIds`) to an existing friends list (`friendsIds`) during an update:

```
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?)`

A helper function that decrements the existing attribute value in the item when updating DynamoDB.

Example

To decrement a friends counter (`friendsCount`) by 10:

```
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?)`

A helper function that increments the existing attribute value in the item when updating DynamoDB.

Example

To increment a friends counter (`friendsCount`) by 10:

```
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)`

A helper function that prepends to the existing list in DynamoDB.

Example

To prepend newly added friend IDs (`newFriendIds`) to an existing friends list (`friendsIds`) during an update:

```
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)`

A helper function that replaces an existing attribute when updating an item in DynamoDB. This is useful for when you want to update the entire object or subobject in the attribute and not just the keys in the payload.

Example

To replace an address (street, city, and zip code) in an `info` object:

```
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)`

A helper function that replaces an item in a list.

Example

In the scope of the update (`newFriendIds`), this example used `updateListItem` to update the ID values of the second item (index: 1, new ID: `102`) and third item (index: 2, new ID: `112`) in a list (`friendsIds`).

```
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 });
}
```

Inputs

Inputs list

`Type GetInput<T>`

```
GetInput<T>: {
    consistentRead?: boolean;
```

```
    key: DynamoDBKey<T>;
}
```

Type Declaration

- `consistentRead?: boolean` (optional)

  An optional boolean to specify whether you want to perform a strongly consistent read with DynamoDB.

- `key: DynamoDBKey<T>` (required)

  A required parameter that specifies the key of the item in DynamoDB. DynamoDB items may have a single hash key or hash and sort keys.

Type PutInput<T>

```
PutInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T> | null;
    customPartitionKey?: string;
    item: Partial<T>;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
}
```

Type Declaration

- `_version?: number` (optional)
- `condition?: DynamoDBFilterObject<T> | null` (optional)

  When you put an object in a DynamoDB table, you can optionally specify a conditional expression that controls whether the request should succeed or not based on the state of the object already in DynamoDB before the operation is performed.

- `customPartitionKey?: string` (optional)

  When enabled, this string value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled. When enabled, the processing of the `populateIndexFields` entry is also enabled.

- `item: Partial<T>` (required)

  The rest of the attributes of the item to be placed into DynamoDB.

- `key: DynamoDBKey<T>` (required)

  A required parameter that specifies the key of the item in DynamoDB on which the put will be performed. DynamoDB items may have a single hash key or hash and sort keys.

- `populateIndexFields?: boolean` (optional)

  A boolean value that, when enabled along with the `customPartitionKey`, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns. For more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*.

`Type QueryInput<T>`

```
QueryInput<T>: ScanInput<T> & {
    query: DynamoDBKeyCondition<Required<T>>;
}
```

Type Declaration

- `query: DynamoDBKeyCondition<Required<T>>` (required)

  Specifies a key condition that describes items to query. For a given index, the condition for a partition key should be an equality and the sort key a comparison or a `beginsWith` (when it's a string). Only number and string types are supported for partition and sort keys.

  Example

  Take the `User` type below:

```
type User = {
  id: string;
  name: string;
  age: number;
  isVerified: boolean;
  friendsIds: string[]
}
```

  The query can only include the following fields: `id`, `name`, and `age`:

```
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;
}
```

Type Declaration

- `_version?: number` (optional)
- `condition?: DynamoDBFilterObject<T>` (optional)

  When you remove an object in DynamoDB, you can optionally specify a conditional expression that controls whether the request should succeed or not based on the state of the object already in DynamoDB before the operation is performed.

  Example

  The following example is a `DeleteItem` expression containing a condition that allows the operation succeed only if the owner of the document matches the user making the request.

```
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` (optional)

  When enabled, the `customPartitionKey` value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled. When enabled, the processing of the `populateIndexFields` entry is also enabled.

- `key: DynamoDBKey<T>` (required)

  A required parameter that specifies the key of the item in DynamoDB that is being removed. DynamoDB items may have a single hash key or hash and sort keys.

  Example

  If a `User` only has the hash key with a user `id`, then the key would look like this:

```
type User = {
 id: number
 name: string
 age: number
 isVerified: boolean
}
const key: DynamoDBKey<User> = {
 id: 1,
}
```

  If the table user has a hash key (`id`) and sort key (`name`), then the key would look like this:

```
type User = {
 id: number
 name: string
 age: number
 isVerified: boolean
 friendsIds: string[]
}

const key: DynamoDBKey<User> = {
 id: 1,
 name: 'XXXXXXXXXX',
}
```

- `populateIndexFields?: boolean` (optional)

A boolean value that, when enabled along with the `customPartitionKey`, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns.

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;
}
```

Type Declaration

- `consistentRead?: boolean | null` (optional)

  An optional boolean to indicate consistent reads when querying DynamoDB. The default value is `false`.

- `filter?: DynamoDBFilterObject<T> | null` (optional)

  An optional filter to apply to the results after retrieving it from the table.

- `index?: string | null` (optional)

  An optional name of the index to scan.

- `limit?: number | null` (optional)

  An optional max number of results to return.

- `nextToken?: string | null` (optional)

  An optional pagination token to continue a previous query. This would have been obtained from a previous query.

- `scanIndexForward?: boolean | null` (optional)

An optional boolean to indicate whether the query is performed in ascending or descending order. By default, this value is set to `true`.

- `segment?: number` (optional)
- `select?: DynamoDBSelectAttributes` (optional)

  Attributes to return from DynamoDB. By default, the AWS AppSync DynamoDB resolver only returns attributes that are projected into the index. The supported values are:

  - `ALL_ATTRIBUTES`

    Returns all the item attributes from the specified table or index. If you query a local secondary index, DynamoDB fetches the entire item from the parent table for each matching item in the index. If the index is configured to project all item attributes, all of the data can be obtained from the local secondary index and no fetching is required.

  - `ALL_PROJECTED_ATTRIBUTES`

    Returns all attributes that have been projected into the index. If the index is configured to project all attributes, this return value is equivalent to specifying `ALL_ATTRIBUTES`.

  - `SPECIFIC_ATTRIBUTES`

    Returns only the attributes listed in `ProjectionExpression`. This return value is equivalent to specifying `ProjectionExpression` without specifying any value for `AttributesToGet`.

- `totalSegments?: number` (optional)

Type `DynamoDBSyncInput<T>`

```
DynamoDBSyncInput<T>: {
    basePartitionKey?: string;
    deltaIndexName?: string;
    filter?: DynamoDBFilterObject<T> | null;
    lastSync?: number;
    limit?: number | null;
    nextToken?: string | null;
}
```

Type Declaration

- `basePartitionKey?: string` (optional)

The partition key of the base table to be used when performing a Sync operation. This field allows a Sync operation to be performed when the table utilizes a custom partition key.

- `deltaIndexName?: string` (optional)

  The index used for the Sync operation. This index is required to enable a Sync operation on the whole delta store table when the table uses a custom partition key. The Sync operation will be performed on the GSI (created on `gsi_ds_pk` and `gsi_ds_sk`).

- `filter?: DynamoDBFilterObject<T> | null` (optional)

  An optional filter to apply to the results after retrieving it from the table.

- `lastSync?: number` (optional)

  The moment, in epoch milliseconds, at which the last successful Sync operation started. If specified, only items that have changed after `lastSync` are returned. This field should only be populated after retrieving all pages from an initial Sync operation. If omitted, results from the base table will be returned. Otherwise, results from the delta table will be returned.

- `limit?: number | null` (optional)

  An optional maximum number of items to evaluate at a single time. If omitted, the default limit will be set to `100` items. The maximum value for this field is `1000` items.

- `nextToken?: string | null` (optional)

Type `DynamoDBUpdateInput<T>`

```
DynamoDBUpdateInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T>;
    customPartitionKey?: string;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
    update: DynamoDBUpdateObject<T>;
}
```

Type Declaration

- `_version?: number` (optional)
- `condition?: DynamoDBFilterObject<T>` (optional)

When you update an object in DynamoDB, you can optionally specify a conditional expression that controls whether the request should succeed or not based on the state of the object already in DynamoDB before the operation is performed.

- `customPartitionKey?: string` (optional)

  When enabled, the `customPartitionKey` value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled. When enabled, the processing of the `populateIndexFields` entry is also enabled.

- `key: DynamoDBKey<T>` (required)

  A required parameter that specifies the key of the item in DynamoDB that is being updated. DynamoDB items may have a single hash key or hash and sort keys.

- `populateIndexFields?: boolean` (optional)

  A boolean value that, when enabled along with the `customPartitionKey`, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns.

- `update: DynamoDBUpdateObject<T>`

  An object that specifies the attributes to be updated along with the new values for them. The update object can be used with `add`, `remove`, `replace`, `increment`, `decrement`, `append`, `prepend`, `updateListItem`.

Amazon RDS module functions

Amazon RDS module functions provide an enhanced experience when interacting with databases configured with the Amazon RDS Data API. The module is imported using `@aws-appsync/utils/rds`:

```
import * as rds from '@aws-appsync/utils/rds';
```

Functions can also be imported individually. For instance, the import below uses `sql`:

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

Functions

You can use the AWS AppSync RDS module's utility helpers to interact with your database.

Select

The `select` utility creates a SELECT statement to query your relational database.

Basic use

In its basic form, you can specify the table you want to query:

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // "SELECT * FROM "persons"
    return createPgStatement(select({table: 'persons'}));
}
```

Note that you can also specify the schema in your table identifier:

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // SELECT * FROM "private"."persons"
    return createPgStatement(select({table: 'private.persons'}));
}
```

Specifying columns

You can specify columns with the `columns` property. If this isn't set to a value, it defaults to *:

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
```

```
        columns: ['id', 'name']
    }));
}
```

You can specify a column's table as well:

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "persons"."name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'persons.name']
    }));
}
```

Limits and offsets

You can apply `limit` and `offset` to the query:

```
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

You can sort your results with the `orderBy` property. Provide an array of objects specifying the column and an optional `dir` property:

```
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'}]
    }));
}
```

Filters

You can build filters by using the special condition object:

```
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'}}
    }));
}
```

You can also combine filters:

```
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}}
    }));
}
```

You can also create OR statements:

```
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 } }
        ]}
    }));
}
```

You can also negate a condition with `not`:

```
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 } }
        ]}
    }));
}
```

You can also use the following operators to compare values:

Operator	Description	Possible value types
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

The `insert` utility provides a straightforward way of inserting single row items in your database with the `INSERT` operation.

Single item insertions

To insert an item, specify the table and then pass in your object of values. The object keys are mapped to your table columns. Columns names are automatically escaped, and values are sent to the database using the variable map:

```
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 use case

You can combine an `insert` followed by a `select` to retrieve your inserted row:

```
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 use case

With Postgres, you can use `returning` to obtain data from the row that you inserted. It accepts * or an array of column names:

```
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)
}
```

Update

The `update` utility allows you to update existing rows. You can use the condition object to apply changes to the specified columns in all the rows that satisfy the condition. For example, let's say we have a schema that allows us to make this mutation. We want to update the `name` of `Person` with the `id` value of `3` but only if we've known them (`known_since`) since the year `2000`:

```
mutation Update {
    updatePerson(
        input: {id: 3, name: "Jon"},
        condition: {known_since: {ge: "2000"}}
    ) {
    id
    name
  }
}
```

Our update resolver looks like this:

```
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)
}
```

We can add a check to our condition to make sure that only the row that has the primary key `id` equal to `3` is updated. Similarly, for Postgres `inserts`, you can use `returning` to return the modified data.

Remove

The `remove` utility allows you to delete existing rows. You can use the condition object on all rows that satisfy the condition. Note that `delete` is a reserved keyword in JavaScript. `remove` should be used instead:

```
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)
}
```

Casting

In some cases, you may want more specificity about the correct object type to use in your statement. You can use the provided type hints to specify the type of your parameters. AWS

AppSync supports the same type hints as the Data API. You can cast your parameters by using the `typeHint` functions from the AWS AppSync `rds` module.

The following example allows you to send an array as a value that is casted as a JSON object. We use the `->` operator to retrieve the element at the `index 2` in the JSON array:

```
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
}
```

Casting is also useful when handling and comparing `DATE`, `TIME`, and `TIMESTAMP`:

```
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)
}
```

Here's another example showing how you can send the current date and time:

```
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)}`)
}
```

Available type hints

- `typeHint.DATE` - The corresponding parameter is sent as an object of the `DATE` type to the database. The accepted format is `YYYY-MM-DD`.
- `typeHint.DECIMAL` - The corresponding parameter is sent as an object of the `DECIMAL` type to the database.
- `typeHint.JSON` - The corresponding parameter is sent as an object of the `JSON` type to the database.
- `typeHint.TIME` - The corresponding string parameter value is sent as an object of the `TIME` type to the database. The accepted format is `HH:MM:SS[.FFF]`.
- `typeHint.TIMESTAMP` - The corresponding string parameter value is sent as an object of the `TIMESTAMP` type to the database. The accepted format is `YYYY-MM-DD HH:MM:SS[.FFF]`.
- `typeHint.UUID` - The corresponding string parameter value is sent as an object of the `UUID` type to the database.

Runtime utilities

The `runtime` library provides utilities to control or modify the runtime properties of your resolvers and functions.

Runtime utils list

`runtime.earlyReturn(obj?: unknown, returnOptions?: {skipTo: 'END' | 'NEXT'}): never`

Invoking this function will halt the execution of the current handler, AWS AppSync function or resolver (Unit or Pipeline Resolver) depending on the current context. The specified object is returned as the result.

- When called in an AWS AppSync function request handler, the data source and response handler are skipped, and the next function request handler (or the pipeline resolver response handler if this was the last AWS AppSync function) is called.
- When called in an AWS AppSync pipeline resolver request handler, the pipeline execution is skipped, and the pipeline resolver response handler is called immediately.
- When `returnOptions` is given with `skipTo` set to "END", the pipeline execution is skipped, and the pipeline resolver response handler is called immediately.
- When `returnOptions` is given with `skipTo` set to "NEXT", the function execution is skipped, and the next pipeline handler is called.

Example

```
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
}
```

Time helpers in util.time

The `util.time` variable contains datetime methods to help generate timestamps, convert between datetime formats, and parse datetime strings. The syntax for datetime formats is based on DateTimeFormatter which you can reference for further documentation.

Time utils list

`util.time.nowISO8601()`

Returns a String representation of UTC in ISO8601 format.

`util.time.nowEpochSeconds()`

Returns the number of seconds from the epoch of 1970-01-01T00:00:00Z to now.

`util.time.nowEpochMilliSeconds()`

Returns the number of milliseconds from the epoch of 1970-01-01T00:00:00Z to now.

`util.time.nowFormatted(String)`

Returns a string of the current timestamp in UTC using the specified format from a String input type.

`util.time.nowFormatted(String, String)`

Returns a string of the current timestamp for a timezone using the specified format and timezone from String input types.

`util.time.parseFormattedToEpochMilliSeconds(String, String)`

Parses a timestamp passed as a String along with a format containing a time zone, then returns the timestamp as milliseconds since epoch.

`util.time.parseFormattedToEpochMilliSeconds(String, String, String)`

Parses a timestamp passed as a String along with a format and time zone, then returns the timestamp as milliseconds since epoch.

`util.time.parseISO8601ToEpochMilliSeconds(String)`

Parses an ISO8601 timestamp passed as a String, then returns the timestamp as milliseconds since epoch.

`util.time.epochMilliSecondsToSeconds(long)`

Converts an epoch milliseconds timestamp to an epoch seconds timestamp.

`util.time.epochMilliSecondsToISO8601(long)`

Converts an epoch milliseconds timestamp to an ISO8601 timestamp.

`util.time.epochMilliSecondsToFormatted(long, String)`

Converts an epoch milliseconds timestamp, passed as long, to a timestamp formatted according to the supplied format in UTC.

`util.time.epochMilliSecondsToFormatted(long, String, String)`

Converts an epoch milliseconds timestamp, passed as a long, to a timestamp formatted according to the supplied format in the supplied timezone.

DynamoDB helpers in util.dynamodb

`util.dynamodb` contains helper methods that make it easier to write and read data to Amazon DynamoDB, such as automatic type mapping and formatting.

toDynamoDB

toDynamoDB utils list

`util.dynamodb.toDynamoDB(Object)`

General object conversion tool for DynamoDB that converts input objects to the appropriate DynamoDB representation. It's opinionated about how it represents some types: e.g., it will use

lists ("L") rather than sets ("SS", "NS", "BS"). This returns an object that describes the DynamoDB attribute value.

String example

```
Input:      util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

Number example

```
Input:      util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

Boolean example

```
Input:      util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

List example

```
Input:      util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

Map example

```
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 list

`util.dynamodb.toString(String)`

Converts an input string to the DynamoDB string format. This returns an object that describes the DynamoDB attribute value.

```
Input:      util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

`util.dynamodb.toStringSet(List<String>)`

Converts a list with Strings to the DynamoDB string set format. This returns an object that describes the DynamoDB attribute value.

```
Input:      util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

toNumber utils

toNumber utils list

`util.dynamodb.toNumber(Number)`

Converts a number to the DynamoDB number format. This returns an object that describes the DynamoDB attribute value.

```
Input:      util.dynamodb.toNumber(12345)
```

```
Output:     { "N" : 12345 }
```

`util.dynamodb.toNumberSet(List<Number>)`

Converts a list of numbers to the DynamoDB number set format. This returns an object that describes the DynamoDB attribute value.

```
Input:      util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

toBinary utils

toBinary utils list

`util.dynamodb.toBinary(String)`

Converts binary data encoded as a base64 string to DynamoDB binary format. This returns an object that describes the DynamoDB attribute value.

```
Input:      util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

`util.dynamodb.toBinarySet(List<String>)`

Converts a list of binary data encoded as base64 strings to DynamoDB binary set format. This returns an object that describes the DynamoDB attribute value.

```
Input:      util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

toBoolean utils

toBoolean utils list

`util.dynamodb.toBoolean(Boolean)`

Converts a Boolean to the appropriate DynamoDB Boolean format. This returns an object that describes the DynamoDB attribute value.

```
Input:      util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

toNull utils

toNull utils list

`util.dynamodb.toNull()`

Returns a null in DynamoDB null format. This returns an object that describes the DynamoDB attribute value.

```
Input:      util.dynamodb.toNull()
Output:     { "NULL" : null }
```

toList utils

toList utils list

`util.dynamodb.toList(List)`

Converts a list of objects to the DynamoDB list format. Each item in the list is also converted to its appropriate DynamoDB format. It's opinionated about how it represents some of the nested objects: e.g., it will use lists ("L") rather than sets ("SS", "NS", "BS"). This returns an object that describes the DynamoDB attribute value.

```
Input:      util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

toMap utils

toMap utils list

`util.dynamodb.toMap(Map)`

Converts a map to the DynamoDB map format. Each value in the map is also converted to its appropriate DynamoDB format. It's opinionated about how it represents some of the nested objects: e.g., it will use lists ("L") rather than sets ("SS", "NS", "BS"). This returns an object that describes the DynamoDB attribute value.

```
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)`

Creates a copy of the map where each value has been converted to its appropriate DynamoDB format. It's opinionated about how it represents some of the nested objects: e.g., it will use lists ("L") rather than sets ("SS", "NS", "BS").

```
Input:      util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "foo"  : { "S" : "bar" },
               "baz"  : { "N" : 1234 },
               "beep" : {
                   "L" : [
                       { "S" : "boop" }
                   ]
               }
           }
```

> **Note**
>
> This is slightly different to `util.dynamodb.toMap(Map)` as it returns only the contents of the DynamoDB attribute value, but not the whole attribute value itself. For example, the following statements are exactly the same:
>
> ```
> util.dynamodb.toMapValues(<map>)
> util.dynamodb.toMap(<map>)("M")
> ```

S3Object utils

S3Object utils list

`util.dynamodb.toS3Object(String key, String bucket, String region)`

Converts the key, bucket and region into the DynamoDB S3 Object representation. This returns an object that describes the DynamoDB attribute value.

```
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)`

Converts the key, bucket, region and optional version into the DynamoDB S3 Object representation. This returns an object that describes the DynamoDB attribute value.

```
Input:      util.dynamodb.toS3Object("foo", "bar", "baz", "beep")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region
\" : \"baz\", \"version\" = \"beep\" } }" }
```

`util.dynamodb.fromS3ObjectJson(String)`

Accepts the string value of a DynamoDB S3 Object and returns a map that contains the key, bucket, region and optional version.

```
Input:      util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
 \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
```

```
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :
 "beep" }
```

HTTP helpers in util.http

The `util.http` utility provides helper methods that you can use to manage HTTP request parameters and to add response headers.

util.http utils list

`util.http.copyHeaders(headers)`

Copies the headers from the map, excluding the following restricted HTTP headers:

- transfer-encoding
- connection
- host
- expect
- keep-alive
- upgrade
- proxy-authenticate
- proxy-authorization
- te
- content-length

`util.http.addResponseHeader(String, Object)`

Adds a single custom header with the name (`String`) and value (`Object`) of the response. The following limitations apply:

- In addition to the list of restricted headers for `copyHeaders(headers)`, header names cannot match any of the following:
  - 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

- Header names can't start with the restricted prefixes `x-amzn-` or `x-amz-`.
- The size of custom response headers can't exceed 4 KB. This includes header names and values.
- You should define each response header once per GraphQL operation. However, if you define a custom header with the same name multiple times, the most recent definition appears in the response. All headers count towards the header size limit regardless of naming.
- Headers with an empty or restricted name (`String`) or a null value (`Object`) will be ignored and yield a `ResponseHeaderError` error that is added to the operation's `errors` output.

```
export function request(ctx) {
  util.http.addResponseHeader('itemsCount', 7)
  util.http.addResponseHeader('render', ctx.args.render)
  return {}
}
```

`util.http.addResponseHeaders(Map)`

Adds multiple response headers to the response from the specified map of names (`String`) and values (`Object`). The same limitations listed for the `addResponseHeader(String, Object)` method also apply to this method.

```
export function request(ctx) {
  const headers = {
    headerInt: 12,
    headerString: 'stringValue',
    headerObject: {
      field1: 7,
      field2: 'string'
    }
  }
  util.http.addResponseHeaders(headers)
  return {}
}
```

Transformation helpers in util.transform

`util.transform` contains helper methods that make it easier to perform complex operations against data sources.

Transformation helpers utils list

`util.transform.toDynamoDBFilterExpression(filterObject: DynamoDBFilterObject) : string`

Converts an input string to a filter expression for use with DynamoDB. We recommend using `toDynamoDBFilterExpression` with built-in module functions.

`util.transform.toElasticsearchQueryDSL(object: OpenSearchQueryObject) : string`

Converts the given input into its equivalent OpenSearch Query DSL expression, returning it as a JSON string.

Example input:

```
util.transform.toElasticsearchQueryDSL({
    "upvotes":{
        "ne":15,
        "range":[
            10,
            20
        ]
    },
    "title":{
        "eq":"hihihi",
        "wildcard":"h*i"
    }
  })
```

Example 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"
                  }
                }
            ]
          }
        }
      ]
    }
}
```

> **Note**
>
> The default operator is assumed to be AND.

`util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter`

Converts a `Map` input object to a `SubscriptionFilter` expression object. The `util.transform.toSubscriptionFilter` method is used as an input to the `extensions.setSubscriptionFilter()` extension. For more information, see Extensions.

> **Note**
>
> The parameters and return statement is listed below:
> *Parameters*
>
> - `objFilter`: `SubscriptionFilterObject`
>
> A `Map` input object that's converted to the `SubscriptionFilter` expression object.
> - `ignoredFields`: `SubscriptionFilterExcludeKeysType` (optional)
>
> A `List` of field names in the first object that will be ignored.
> - `rules`: `SubscriptionFilterRuleObject` (optional)
>
> A `Map` input object with strict rules that's included when you're constructing the `SubscriptionFilter` expression object. These strict rules will be included in the `SubscriptionFilter` expression object so that at least one of the rules will be satisfied to pass the subscription filter.
>
> *Response*
> Returns a `SubscriptionFilter`.

`util.transform.toSubscriptionFilter(Map, List)`

Converts a `Map` input object to a `SubscriptionFilter` expression object. The `util.transform.toSubscriptionFilter` method is used as an input to the `extensions.setSubscriptionFilter()` extension. For more information, see Extensions.

The first argument is the `Map` input object that's converted to the `SubscriptionFilter` expression object. The second argument is a `List` of field names that are ignored in the first `Map` input object while constructing the `SubscriptionFilter` expression object.

`util.transform.toSubscriptionFilter(Map, List, Map)`

Converts a `Map` input object to a `SubscriptionFilter` expression object. The `util.transform.toSubscriptionFilter` method is used as an input to the `extensions.setSubscriptionFilter()` extension. For more information, see Extensions.

`util.transform.toDynamoDBConditionExpression(conditionObject)`

Creates a DynamoDB condition expression.

Subscription filter arguments

The following table explains the how the arguments of the following utilities are defined:

- `Util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter`

Argument 1: Map

Argument 1 is a `Map` object with the following key values:

- field names
- "and"
- "or"

For field names as keys, the conditions on these fields' entries are in the form of `"operator" : "value"`.

The following example shows how entries can be added to the `Map`:

```
"field_name" : {
                    "operator1" : value
                }

## We can have multiple conditions for the same field_name:

"field_name" : {
```

```
                    "operator1" : value
                    "operator2" : value
                    .
                    .
                    .
                }
```

When a field has two or more conditions on it, all of these conditions are considered to use the OR operation.

The input `Map` can also have "and" and "or" as keys, implying that all entries within these should be joined using AND or OR logic depending on the key. The key values "and" and "or" expect an array of conditions.

```
"and" : [

            {
                "field_name1" : {
                    "operator1" : value
                }
             },

             {
                "field_name2" : {
                    "operator1" : value
                }
             },
             .
             .
        ].
```

Note that you can nest "and" and "or". That is, you can have nested "and"/"or" within another "and"/"or" block. However, this doesn't work for simple fields.

```
"and" : [

            {
                "field_name1" : {
                    "operator" : value
                }
             },
```

```
            {
               "or" : [
                           {
                               "field_name2" : {
                                   "operator" : value
                               }
                           },

                           {
                               "field_name3" : {
                                   "operator" : value
                               }
                           }

                       ].
```

The following example shows an input of *argument 1* using `util.transform.toSubscriptionFilter(Map) : Map`.

Input(s)

Argument 1: Map:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
```

```
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Output

The result is a `Map` object:

```
{
  "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

Argument 2 contains a `List` of field names that shouldn't be considered in the input `Map` (argument 1) while constructing the `SubscriptionFilter` expression object. The `List` can also be empty.

The following example shows the inputs of argument 1 and argument 2 using `util.transform.toSubscriptionFilter(Map, List) : Map`.

Input(s)

Argument 1: Map:

```
{
  "percentageUp": {
```

```
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argument 2: List:

```
["percentageUp", "author"]
```

Output

The result is a `Map` object:

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
```

```
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 3: Map

Argument 3 is a `Map` object that has field names as key values (cannot have "and" or "or"). For field names as keys, the conditions on these fields are entries in the form of `"operator" : "value"`. Unlike argument 1, argument 3 cannot have multiple conditions in the same key. In addition, argument 3 doesn't have an "and" or "or" clause, so there's no nesting involved either.

Argument 3 represents a list of strict rules, which are added to the `SubscriptionFilter` expression object so that **at least one** of these conditions is met to pass the filter.

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.
```

The following example shows the inputs of *argument 1*, *argument 2*, and *argument 3* using `util.transform.toSubscriptionFilter(Map, List, Map) : Map`.

Input(s)

Argument 1: Map:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argument 2: List:

```
["percentageUp", "author"]
```

Argument 3: Map:

```
{
```

```
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
}
```

Output

The result is a Map object:

```
{
  "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"
        }
      ]
    }
  ]
}
```

String helpers in util.str

`util.str` contains methods to help with common String operations.

util.str utils list

`util.str.normalize(String, String)`

Normalizes a string using one of the four unicode normalization forms: NFC, NFD, NFKC, or NFKD. The first argument is the string to normalize. The second argument is either "nfc", "nfd", "nfkc", or "nfkd" specifying the normalization type to use for the normalization process.

Extensions

`extensions` contains a set of methods to make additional actions within your resolvers.

Caching extensions

`extensions.evictFromApiCache(typeName: string, fieldName: string, keyValuePair: Record<string, string>) : Object`

Evicts an item from the AWS AppSync server-side cache. The first argument is the type name. The second argument is the field name. The third argument is an object containing key-value pair items that specify the caching key value. You must put the items in the object in the same order as the caching keys in the cached resolver's `cachingKey`. For more information about caching, see Caching behavior.

Example 1:

This example evicts the items that were cached for a resolver called `Query.allClasses` on which a caching key called `context.arguments.semester` was used. When the mutation is called and the resolver runs, if an entry is successfully cleared, then the response contains an `apiCacheEntriesDeleted` value in the extensions object that shows how many entries were deleted.

```
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**
>
> This function **only** works for mutations, not queries.

Subscription extensions

`extensions.setSubscriptionFilter(filterJsonObject)`

Defines enhanced subscription filters. Each subscription notification event is evaluated against provided subscription filters and delivers notifications to clients if all filters evaluate to `true`.

The argument is `filterJsonObject` (More information about this argument can be found below in the *Argument: filterJsonObject* section.). See Enhanced subscription filtering.

> **Note**
>
> You can use this extension function only in the response handler of a subscription resolver. Also, we recommend using `util.transform.toSubscriptionFilter` to create your filter.

`extensions.setSubscriptionInvalidationFilter(filterJsonObject)`

Defines subscription invalidation filters. Subscription filters are evaluated against the invalidation payload, then invalidate a given subscription if the filters evaluate to `true`. The argument is `filterJsonObject` (More information about this argument can be found below in the *Argument: filterJsonObject* section.). See Enhanced subscription filtering.

> **Note**
>
> You can use this extension function only in the response handler of a subscription resolver. Also, we recommend using `util.transform.toSubscriptionFilter` to create your filter.

`extensions.invalidateSubscriptions(invalidationJsonObject)`

Used to initiate a subscription invalidation from a mutation. The argument is `invalidationJsonObject` (More information about this argument can be found below in the *Argument: invalidationJsonObject* section.).

> **Note**
>
> This extension can be used only in the response mapping templates of the mutation resolvers.
> You can only use at most five unique `extensions.invalidateSubscriptions()` method calls in any single request. If you exceed this limit, you will receive a GraphQL error.

Argument: filterJsonObject

The JSON object defines either subscription or invalidation filters. It's an array of filters in a `filterGroup`. Each filter is a collection of individual filters.

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

Each filter has three attributes:

- `fieldName` – The GraphQL schema field.
- `operator` – The operator type.
- `value` – The values to compare to the subscription notification `fieldName` value.

The following is an example assignment of these attributes:

```
{
 "fieldName" : "severity",
 "operator" : "le",
```

```
 "value" : context.result.severity
}
```

Argument: invalidationJsonObject

The `invalidationJsonObject` defines the following:

- `subscriptionField` – The GraphQL schema subscription to invalidate. A single subscription, defined as a string in the `subscriptionField`, is considered for invalidation.
- `payload` – A key-value pair list that's used as the input for invalidating subscriptions if the invalidation filter evaluates to `true` against their values.

  The following example invalidates subscribed and connected clients using the `onUserDelete` subscription when the invalidation filter defined in the subscription resolver evaluates to `true` against the `payload` value.

```
export const request = (ctx) => ({ payload: null });

export function response(ctx) {
 extensions.invalidateSubscriptions({
  subscriptionField: 'onUserDelete',
  payload: { group: 'Developer', type: 'Full-Time' },
 });
 return ctx.result;
}
```

XML helpers in util.xml

`util.xml` contains methods to help with XML string conversion.

util.xml utils list

`util.xml.toMap(String) : Object`

Converts a XML string to a dictionary.

Example 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"
      }
    }
}
```

Example 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`

Converts a XML string to a JSON string. This is similar to `toMap`, except that the output is a string. This is useful if you want to directly convert and return the XML response from an HTTP object to JSON. You can set an optional boolean parameter to determine if you want to string-encode the JSON.

AWS AppSync JavaScript resolver function reference for DynamoDB

The AWS AppSync DynamoDB function allows you to use GraphQL to store and retrieve data in existing Amazon DynamoDB tables in your account by mapping an incoming GraphQL request into a DynamoDB call, and then mapping the DynamoDB response back to GraphQL. This section describes the request and response handlers for supported DynamoDB operations:

- GetItem - The GetItem request lets you tell the DynamoDB function to make a GetItem request to DynamoDB, and enables you to specify the key of the item in DynamoDB and whether to use a consistent read or not.
- PutItem - The PutItem request mapping document lets you tell the DynamoDB function to make a PutItem request to DynamoDB, and enables you to specify the key of the item in DynamoDB, the full contents of the item (composed of key and attributeValues), and conditions for the operation to succeed.
- UpdateItem - The UpdateItem request enables you to tell the DynamoDB function to make a UpdateItem request to DynamoDB and allows you to specify the key of the item in DynamoDB, an update expression describing how to update the item in DynamoDB, and conditions for the operation to succeed.

- DeleteItem - The DeleteItem request lets you tell the DynamoDB function to make a DeleteItem request to DynamoDB, and enables you to specify the key of the item in DynamoDB and conditions for the operation to succeed.
- Query - The Query request object lets you tell the DynamoDB resolver to make a Query request to DynamoDB, and enables you to specify the key expression, which index to use, additional filters, how many items to return, whether to use consistent reads, query direction (forward or backward), and pagination tokens.
- Scan - The Scan request lets you tell the DynamoDB function to make a Scan request to DynamoDB, and enables you to specify a filter to exclude results, which index to use, how many items to return, whether to use consistent reads, pagination tokens, and parallel scans.
- Sync - The Sync request object lets you retrieve all the results from a DynamoDB table and then receive only the data altered since your last query (the delta updates). Sync requests can only be made to versioned DynamoDB data sources. You can specify a filter to exclude results, how many items to return, pagination Tokens, and when your last Sync operation was started.
- BatchGetItem - The BatchGetItem request object lets you tell the DynamoDB function to make a BatchGetItem request to DynamoDB to retrieve multiple items, potentially across multiple tables. For this request object, you must specify the table names to retrieve the items from and the keys of the items to retrieve from each table.
- BatchDeleteItem - The BatchDeleteItem request object lets you tell the DynamoDB function to make a BatchWriteItem request to DynamoDB to delete multiple items, potentially across multiple tables. For this request object, you must specify the table names to delete the items from and the keys of the items to delete from each table.
- BatchPutItem - The BatchPutItem request object lets you tell the DynamoDB function to make a BatchWriteItem request to DynamoDB to put multiple items, potentially across multiple tables. For this request object, you must specify the table names to put the items in and the full items to put in each table.
- TransactGetItems - The TransactGetItems request object lets you to tell the DynamoDB function to make a TransactGetItems request to DynamoDB to retrieve multiple items, potentially across multiple tables. For this request object, you must specify the table name of each request item to retrieve the item from and the key of each request item to retrieve from each table.
- TransactWriteItems - The TransactWriteItems request object lets you tell the DynamoDB function to make a TransactWriteItems request to DynamoDB to write multiple items, potentially to multiple tables. For this request object, you must specify the destination table name of each request item, the operation of each request item to perform, and the key of each request item to write.

- Type system (request mapping) - Learn more about how DynamoDB typing is integrated into AWS AppSync requests.
- Type system (response mapping) - Learn more about how DynamoDB types are converted automatically to GraphQL or JSON in a response payload.
- Filters - Learn more about filters for query and scan operations.
- Condition expressions - Learn more about condition expressions for PutItem, UpdateItem, and DeleteItem operations.
- Transaction condition expressions - Learn more about condition expressions for TransactWriteItems operations.
- Projections - Learn more about how to specify attributes in read operations.

GetItem

The `GetItem` request lets you tell the AWS AppSync DynamoDB function to make a `GetItem` request to DynamoDB, and enables you to specify:

- The key of the item in DynamoDB
- Whether to use a consistent read or not

The `GetItem` request has the following structure:

```
type DynamoDBGetItem = {
  operation: 'GetItem';
  key: { [key: string]: any };
  consistentRead?: ConsistentRead;
  projection?: {
    expression: string;
    expressionNames?: { [key: string]: string };
  };
};
```

The fields are defined as follows:

GetItem fields

GetItem fields list

`operation`

The DynamoDB operation to perform. To perform the `GetItem` DynamoDB operation, this must be set to `GetItem`. This value is required.

`key`

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a "typed value", see Type system (request mapping). This value is required.

`consistentRead`

Whether or not to perform a strongly consistent read with DynamoDB. This is optional, and defaults to `false`.

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

The item returned from DynamoDB is automatically converted into GraphQL and JSON primitive types, and is available in the context result (`context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about JavaScript resolvers, see JavaScript resolvers overview.

Example

The following example is a function request handler for a GraphQL query `getThing(foo: String!, bar: String!)`:

```
export function request(ctx) {
  const {foo, bar} = ctx.args
  return {
    operation : "GetItem",
    key : util.dynamodb.toMapValues({foo, bar}),
    consistentRead : true
  }
```

```
}
```

For more information about the DynamoDB `GetItem` API, see the DynamoDB API documentation.

PutItem

The `PutItem` request mapping document lets you tell the AWS AppSync DynamoDB function to make a `PutItem` request to DynamoDB, and enables you to specify the following:

- The key of the item in DynamoDB
- The full contents of the item (composed of `key` and `attributeValues`)
- Conditions for the operation to succeed

The `PutItem` request has the following structure:

```
type DynamoDBPutItemRequest = {
  operation: 'PutItem';
  key: { [key: string]: any };
  attributeValues: { [key: string]: any};
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

The fields are defined as follows:

PutItem fields

PutItem fields list

`operation`

The DynamoDB operation to perform. To perform the `PutItem` DynamoDB operation, this must be set to `PutItem`. This value is required.

`key`

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a "typed value", see Type system (request mapping). This value is required.

`attributeValues`

The rest of the attributes of the item to be put into DynamoDB. For more information about how to specify a “typed value”, see Type system (request mapping). This field is optional.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `PutItem` request overwrites any existing entry for that item. For more information about conditions, see Condition expressions. This value is optional.

`_version`

A numeric value that represents the latest known version of an item. This value is optional. This field is used for *Conflict Detection* and is only supported on versioned data sources.

`customPartitionKey`

When enabled, this string value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled (for more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*). When enabled, the processing of the `populateIndexFields` entry is also enabled. This field is optional.

`populateIndexFields`

A boolean value that, when enabled **along with the `customPartitionKey`**, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns. For more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*. This field is optional.

The item written to DynamoDB is automatically converted to GraphQL and JSON primitive types and is available in the context result (`context.result`).

The item written to DynamoDB is automatically converted into GraphQL and JSON primitive types and is available in the context result (`context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about JavaScript resolvers, see JavaScript resolvers overview.

Example 1

The following example is a function request handler for a GraphQL mutation `updateThing(foo: String!, bar: String!, name: String!, version: Int!)`.

If no item with the specified key exists, it's created. If an item already exists with the specified key, it's overwritten.

```
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),
  };
}
```

Example 2

The following example is a function request handler for a GraphQL mutation `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)`.

This example verifies that the item currently in DynamoDB has the `version` field set to `expectedVersion`.

```
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,
  };
}
```

For more information about the DynamoDB `PutItem` API, see the DynamoDB API documentation.

UpdateItem

The `UpdateItem` request enables you to tell the AWS AppSync DynamoDB function to make a `UpdateItem` request to DynamoDB and allows you to specify the following:

- The key of the item in DynamoDB
- An update expression describing how to update the item in DynamoDB
- Conditions for the operation to succeed

The `UpdateItem` request has the following structure:

```
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;
};
```

The fields are defined as follows:

UpdateItem fields

UpdateItem fields list

`operation`

The DynamoDB operation to perform. To perform the `UpdateItem` DynamoDB operation, this must be set to `UpdateItem`. This value is required.

key

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about specifying a "typed value", see Type system (request mapping). This value is required.

update

The `update` section lets you specify an update expression that describes how to update the item in DynamoDB. For more information about how to write update expressions, see the DynamoDB UpdateExpressions documentation. This section is required.

The `update` section has three components:

expression

The update expression. This value is required.

expressionNames

The substitutions for expression attribute *name* placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the `expression`, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the `expression`.

expressionValues

The substitutions for expression attribute *value* placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the `expression`, and the value must be a typed value. For more information about how to specify a "typed value", see Type system (request mapping). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the `expression`.

condition

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `UpdateItem` request updates the existing entry regardless of its current state. For more information about conditions, see Condition expressions. This value is optional.

`_version`

A numeric value that represents the latest known version of an item. This value is optional. This field is used for *Conflict Detection* and is only supported on versioned data sources.

`customPartitionKey`

When enabled, this string value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled (for more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*). When enabled, the processing of the `populateIndexFields` entry is also enabled. This field is optional.

`populateIndexFields`

A boolean value that, when enabled **along with the `customPartitionKey`**, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns. For more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*. This field is optional.

The item updated in DynamoDB is automatically converted into GraphQL and JSON primitive types and is available in the context result (`context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about JavaScript resolvers, see JavaScript resolvers overview.

Example 1

The following example is a function request handler for the GraphQL mutation `upvote(id: ID!)`.

In this example, an item in DynamoDB has its `upvotes` and `version` fields incremented by 1.

```
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 } },
    },
  };
}
```

Example 2

The following example is a function request handler for a GraphQL mutation `updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)`.

This is a complex example that inspects the arguments and dynamically generates the update expression that only includes the arguments that have been provided by the client. For example, if `title` and `author` are omitted, they are not updated. If an argument is specified but its value is `null`, then that field is deleted from the object in DynamoDB. Finally, the operation has a condition, which verifies whether the item currently in DynamoDB has the `version` field set to `expectedVersion`:

```
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),
    },
  };
}
```

For more information about the DynamoDB `UpdateItem` API, see the DynamoDB API documentation.

DeleteItem

The `DeleteItem` request lets you tell the AWS AppSync DynamoDB function to make a `DeleteItem` request to DynamoDB, and enables you to specify the following:

- The key of the item in DynamoDB
- Conditions for the operation to succeed

The `DeleteItem` request has the following structure:

```
type DynamoDBDeleteItemRequest = {
  operation: 'DeleteItem';
  key: { [key: string]: any };
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

The fields are defined as follows:

DeleteItem fields

DeleteItem fields list

`operation`

The DynamoDB operation to perform. To perform the `DeleteItem` DynamoDB operation, this must be set to `DeleteItem`. This value is required.

`key`

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about specifying a "typed value", see Type system (request mapping). This value is required.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `DeleteItem` request deletes an item regardless of its current state. For more information about conditions, see Condition expressions. This value is optional.

`_version`

A numeric value that represents the latest known version of an item. This value is optional. This field is used for *Conflict Detection* and is only supported on versioned data sources.

`customPartitionKey`

When enabled, this string value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled (for more information, see Conflict

detection and sync in the *AWS AppSync Developer Guide*). When enabled, the processing of the `populateIndexFields` entry is also enabled. This field is optional.

`populateIndexFields`

A boolean value that, when enabled **along with the `customPartitionKey`**, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns. For more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*. This field is optional.

The item deleted from DynamoDB is automatically converted into GraphQL and JSON primitive types and is available in the context result (`context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about JavaScript resolvers, see JavaScript resolvers overview.

Example 1

The following example is a function request handler for a GraphQL mutation `deleteItem(id: ID!)`. If an item exists with this ID, it's deleted.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'DeleteItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}
```

Example 2

The following example is a function request handler for a GraphQL mutation `deleteItem(id: ID!, expectedVersion: Int!)`. If an item exists with this ID, it's deleted, but only if its `version` field set to `expectedVersion`:

```
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),
  };
}
```

For more information about the DynamoDB `DeleteItem` API, see the DynamoDB API documentation.

Query

The `Query` request object lets you tell the AWS AppSync DynamoDB resolver to make a `Query` request to DynamoDB, and enables you to specify the following:

- Key expression
- Which index to use
- Any additional filter
- How many items to return
- Whether to use consistent reads
- query direction (forward or backward)
- Pagination token

The `Query` request object has the following structure:

```
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 };
  };
};
```

The fields are defined as follows:

Query fields

Query fields list

`operation`

The DynamoDB operation to perform. To perform the `Query` DynamoDB operation, this must be set to `Query`. This value is required.

`query`

The `query` section lets you specify a key condition expression that describes which items to retrieve from DynamoDB. For more information about how to write key condition expressions, see the DynamoDB KeyConditions documentation . This section must be specified.

`expression`

The query expression. This field must be specified.

`expressionNames`

The substitutions for expression attribute *name* placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the `expression`, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the `expression`.

`expressionValues`

The substitutions for expression attribute *value* placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the `expression`, and the value must be a typed value. For more information about how to specify a "typed value", see Type system (request mapping). This value is required. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the `expression`.

`filter`

An additional filter that can be used to filter the results from DynamoDB before they are returned. For more information about filters, see Filters. This field is optional.

`index`

The name of the index to query. The DynamoDB query operation allows you to scan on Local Secondary Indexes and Global Secondary Indexes in addition to the primary key index for a hash key. If specified, this tells DynamoDB to query the specified index. If omitted, the primary key index is queried.

`nextToken`

The pagination token to continue a previous query. This would have been obtained from a previous query. This field is optional.

`limit`

The maximum number of items to evaluate (not necessarily the number of matching items). This field is optional.

`scanIndexForward`

A boolean indicating whether to query forwards or backwards. This field is optional, and defaults to `true`.

`consistentRead`

A boolean indicating whether to use consistent reads when querying DynamoDB. This field is optional, and defaults to `false`.

`select`

By default, the AWS AppSync DynamoDB resolver only returns attributes that are projected into the index. If more attributes are required, you can set this field. This field is optional. The supported values are:

`ALL_ATTRIBUTES`

Returns all of the item attributes from the specified table or index. If you query a local secondary index, DynamoDB fetches the entire item from the parent table for each matching item in the index. If the index is configured to project all item attributes, all of the data can be obtained from the local secondary index and no fetching is required.

`ALL_PROJECTED_ATTRIBUTES`

Allowed only when querying an index. Retrieves all attributes that have been projected into the index. If the index is configured to project all attributes, this return value is equivalent to specifying `ALL_ATTRIBUTES`.

`SPECIFIC_ATTRIBUTES`

Returns only the attributes listed in the `projection`'s `expression`. This return value is equivalent to specifying the `projection`'s `expression` without specifying any value for `Select`.

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

The results from DynamoDB are automatically converted into GraphQL and JSON primitive types and are available in the context result (`context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about JavaScript resolvers, see JavaScript resolvers overview.

The results have the following structure:

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

The fields are defined as follows:

`items`

A list containing the items returned by the DynamoDB query.

`nextToken`

If there might be more results, `nextToken` contains a pagination token that you can use in another request. Note that AWS AppSync encrypts and obfuscates the pagination token returned from DynamoDB. This prevents your table data from being inadvertently leaked to the caller. Also note that these pagination tokens cannot be used across different functions or resolvers.

`scannedCount`

The number of items that matched the query condition expression, before a filter expression (if present) was applied.

Example

The following example is a function request handler for a GraphQL query `getPosts(owner: ID!)`.

In this example, a global secondary index on a table is queried to return all posts owned by the specified ID.

```
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',
  };
}
```

For more information about the DynamoDB `Query` API, see the DynamoDB API documentation.

Scan

The `Scan` request lets you tell the AWS AppSync DynamoDB function to make a `Scan` request to DynamoDB, and enables you to specify the following:

- A filter to exclude results
- Which index to use
- How many items to return
- Whether to use consistent reads
- Pagination token
- Parallel scans

The `Scan` request object has the following structure:

```
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 };
  };
};
```

The fields are defined as follows:

Scan fields

Scan fields list

`operation`

The DynamoDB operation to perform. To perform the `Scan` DynamoDB operation, this must be set to `Scan`. This value is required.

`filter`

A filter that can be used to filter the results from DynamoDB before they are returned. For more information about filters, see Filters. This field is optional.

`index`

The name of the index to query. The DynamoDB query operation allows you to scan on Local Secondary Indexes and Global Secondary Indexes in addition to the primary key index for a hash key. If specified, this tells DynamoDB to query the specified index. If omitted, the primary key index is queried.

`limit`

The maximum number of items to evaluate at a single time. This field is optional.

`consistentRead`

A Boolean that indicates whether to use consistent reads when querying DynamoDB. This field is optional, and defaults to `false`.

`nextToken`

The pagination token to continue a previous query. This would have been obtained from a previous query. This field is optional.

`select`

By default, the AWS AppSync DynamoDB function only returns whatever attributes are projected into the index. If more attributes are required, then this field can be set. This field is optional. The supported values are:

`ALL_ATTRIBUTES`

Returns all of the item attributes from the specified table or index. If you query a local secondary index, DynamoDB fetches the entire item from the parent table for each

matching item in the index. If the index is configured to project all item attributes, all of the data can be obtained from the local secondary index and no fetching is required.

`ALL_PROJECTED_ATTRIBUTES`

Allowed only when querying an index. Retrieves all attributes that have been projected into the index. If the index is configured to project all attributes, this return value is equivalent to specifying `ALL_ATTRIBUTES`.

`SPECIFIC_ATTRIBUTES`

Returns only the attributes listed in the `projection`'s `expression`. This return value is equivalent to specifying the `projection`'s `expression` without specifying any value for `Select`.

`totalSegments`

The number of segments to partition the table by when performing a parallel scan. This field is optional, but must be specified if `segment` is specified.

`segment`

The table segment in this operation when performing a parallel scan. This field is optional, but must be specified if `totalSegments` is specified.

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

The results returned by the DynamoDB scan are automatically converted into GraphQL and JSON primitive types and is available in the context result (`context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about JavaScript resolvers, see JavaScript resolvers overview.

The results have the following structure:

```
{
    items = [ ... ],
    nextToken = "a pagination token",
```

```
    scannedCount = 10
}
```

The fields are defined as follows:

`items`

A list containing the items returned by the DynamoDB scan.

`nextToken`

If there might be more results, `nextToken` contains a pagination token that you can use in another request. AWS AppSync encrypts and obfuscates the pagination token returned from DynamoDB. This prevents your table data from being inadvertently leaked to the caller. Also, these pagination tokens can't be used across different functions or resolvers.

`scannedCount`

The number of items that were retrieved by DynamoDB before a filter expression (if present) was applied.

Example 1

The following example is a function request handler for the GraphQL query: `allPosts`.

In this example, all entries in the table are returned.

```
export function request(ctx) {
  return { operation: 'Scan' };
}
```

Example 2

The following example is a function request handler for the GraphQL query: `postsMatching(title: String!)`.

In this example, all entries in the table are returned where the title starts with the `title` argument.

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

```
  const { title } = ctx.args;
  const filter = { filter: { beginsWith: title } };
  return {
    operation: 'Scan',
    filter: JSON.parse(util.transform.toDynamoDBFilterExpression(filter)),
  };
}
```

For more information about the DynamoDB `Scan` API, see the DynamoDB API documentation.

Sync

The `Sync` request object lets you retrieve all the results from a DynamoDB table and then receive only the data altered since your last query (the delta updates). `Sync` requests can only be made to versioned DynamoDB data sources. You can specify the following:

- A filter to exclude results
- How many items to return
- Pagination Token
- When your last `Sync` operation was started

The `Sync` request object has the following structure:

```
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 };
  };
};
```

The fields are defined as follows:

Sync fields

Sync fields list

`operation`

The DynamoDB operation to perform. To perform the `Sync` operation, this must be set to `Sync`. This value is required.

`filter`

A filter that can be used to filter the results from DynamoDB before they are returned. For more information about filters, see Filters. This field is optional.

`limit`

The maximum number of items to evaluate at a single time. This field is optional. If omitted, the default limit will be set to `100` items. The maximum value for this field is `1000` items.

`nextToken`

The pagination token to continue a previous query. This would have been obtained from a previous query. This field is optional.

`lastSync`

The moment, in epoch milliseconds, when the last successful `Sync` operation started. If specified, only items that have changed after `lastSync` are returned. This field is optional, and should only be populated after retrieving all pages from an initial `Sync` operation. If omitted, results from the *Base* table will be returned, otherwise, results from the *Delta* table will be returned.

`basePartitionKey`

The partition key of the *Base* table used when performing a `Sync` operation. This field allows a `Sync` operation to be performed when the table utilizes a custom partition key. This is an optional field.

`deltaIndexName`

The index used for the `Sync` operation. This index is required to enable a `Sync` operation on the whole delta store table when the table uses a custom partition key. The `Sync` operation will be performed on the GSI (created on `gsi_ds_pk` and `gsi_ds_sk`). This field is optional.

The results returned by the DynamoDB sync are automatically converted into GraphQL and JSON primitive types and are available in the context result (`context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about JavaScript resolvers, see JavaScript resolvers overview.

The results have the following structure:

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10,
    startedAt = 1550000000000
}
```

The fields are defined as follows:

`items`

A list containing the items returned by the sync.

`nextToken`

If there might be more results, `nextToken` contains a pagination token that you can use in another request. AWS AppSync encrypts and obfuscates the pagination token returned from DynamoDB. This prevents your table data from being inadvertently leaked to the caller. Also, these pagination tokens can't be used across different functions or resolvers.

`scannedCount`

The number of items that were retrieved by DynamoDB before a filter expression (if present) was applied.

`startedAt`

The moment, in epoch milliseconds, when the sync operation started that you can store locally and use in another request as your `lastSync` argument. If a pagination token was included in the request, this value will be the same as the one returned by the request for the first page of results.

Example

The following example is a function request handler for the GraphQL query: `syncPosts(nextToken: String, lastSync: AWSTimestamp)`.

In this example, if `lastSync` is omitted, all entries in the base table are returned. If `lastSync` is supplied, only the entries in the delta sync table that have changed since `lastSync` are returned.

```
export function request(ctx) {
  const { nextToken, lastSync } = ctx.args;
  return { operation: 'Sync', limit: 100, nextToken, lastSync };
}
```

BatchGetItem

The `BatchGetItem` request object lets you tell the AWS AppSync DynamoDB function to make a `BatchGetItem` request to DynamoDB to retrieve multiple items, potentially across multiple tables. For this request object, you must specify the following:

- The table names where to retrieve the items from
- The keys of the items to retrieve from each table

The DynamoDB `BatchGetItem` limits apply and **no condition expression** can be provided.

The `BatchGetItem` request object has the following structure:

```
type DynamoDBBatchGetItemRequest = {
  operation: 'BatchGetItem';
  tables: {
    [tableName: string]: {
      keys: { [key: string]: any }[];
      consistentRead?: boolean;
      projection?: {
        expression: string;
        expressionNames?: { [key: string]: string };
      };
    };
  };
};
```

The fields are defined as follows:

BatchGetItem fields

BatchGetItem fields list

`operation`

The DynamoDB operation to perform. To perform the `BatchGetItem` DynamoDB operation, this must be set to `BatchGetItem`. This value is required.

`tables`

The DynamoDB tables to retrieve the items from. The value is a map where table names are specified as the keys of the map. At least one table must be provided. This `tables` value is required.

`keys`

List of DynamoDB keys representing the primary key of the items to retrieve. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping).

`consistentRead`

Whether to use a consistent read when executing a *GetItem* operation. This value is optional and defaults to *false*.

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

Things to remember:

- If an item has not been retrieved from the table, a *null* element appears in the data block for that table.
- Invocation results are sorted per table, based on the order in which they were provided inside the request object.
- Each `Get` command inside a `BatchGetItem` is atomic, however, a batch can be partially processed. If a batch is partially processed due to an error, the unprocessed keys are returned as part of the invocation result inside the *unprocessedKeys* block.

- `BatchGetItem` is limited to 100 keys.

For the following example function request handler:

```
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 })],
    },
  };
}
```

The invocation result available in `ctx.result` is as follows:

```
{
   "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": []
   }
}
```

The `ctx.error` contains details about the error. The keys **data**, **unprocessedKeys**, and each table key that was provided in the result in the function request object are guaranteed to be present in the invocation result. Items that have been deleted appear in the **data** block. Items that haven't been processed are marked as *null* inside the data block and are placed inside the **unprocessedKeys** block.

BatchDeleteItem

The `BatchDeleteItem` request object lets you tell the AWS AppSync DynamoDB function to make a `BatchWriteItem` request to DynamoDB to delete multiple items, potentially across multiple tables. For this request object, you must specify the following:

- The table names where to delete the items from
- The keys of the items to delete from each table

The DynamoDB `BatchWriteItem` limits apply and **no condition expression** can be provided.

The `BatchDeleteItem` request object has the following structure:

```
type DynamoDBBatchDeleteItemRequest = {
  operation: 'BatchDeleteItem';
  tables: {
    [tableName: string]: { [key: string]: any }[];
  };
};
```

The fields are defined as follows:

BatchDeleteItem fields

BatchDeleteItem fields list

`operation`

The DynamoDB operation to perform. To perform the `BatchDeleteItem` DynamoDB operation, this must be set to `BatchDeleteItem`. This value is required.

`tables`

The DynamoDB tables to delete the items from. Each table is a list of DynamoDB keys representing the primary key of the items to delete. DynamoDB items may have a single hash

key, or a hash key and sort key, depending on the table structure. For more information about how to specify a "typed value", see Type system (request mapping). At least one table must be provided. The `tables` value is required.

Things to remember:

- Contrary to the `DeleteItem` operation, the fully deleted item isn't returned in the response. Only the passed key is returned.
- If an item has not been deleted from the table, a *null* element appears in the data block for that table.
- Invocation results are sorted per table, based on the order in which they were provided inside the request object.
- Each `Delete` command inside a `BatchDeleteItem` is atomic. However a batch can be partially processed. If a batch is partially processed due to an error, the unprocessed keys are returned as part of the invocation result inside the *unprocessedKeys* block.
- `BatchDeleteItem` is limited to 25 keys.
- This operation **is not** supported when used with conflict detection. Using both at the same time may result in an error.

For the following example function request handler:

```
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 })],
    },
  };
}
```

The invocation result available in `ctx.result` is as follows:

```
{
```

```
   "data": {
     "authors": [null],
     "posts": [
        // Was deleted
        {
          "authorId": "a1",
          "postId": "p2"
        }
     ]
   },
   "unprocessedKeys": {
     "authors": [
        // This key was not processed due to an error
        {
          "authorId": "a1"
        }
      ],
     "posts": []
   }
}
```

The `ctx.error` contains details about the error. The keys **data**, **unprocessedKeys**, and each table key that was provided in the function request object are guaranteed to be present in the invocation result. Items that have been deleted are present in the **data** block. Items that haven't been processed are marked as *null* inside the data block and are placed inside the **unprocessedKeys** block.

BatchPutItem

The `BatchPutItem` request object lets you tell the AWS AppSync DynamoDB function to make a `BatchWriteItem` request to DynamoDB to put multiple items, potentially across multiple tables. For this request object, you must specify the following:

- The table names where to put the items in
- The full items to put in each table

The DynamoDB `BatchWriteItem` limits apply and **no condition expression** can be provided.

The `BatchPutItem` request object has the following structure:

```
type DynamoDBBatchPutItemRequest = {
```

```
  operation: 'BatchPutItem';
  tables: {
    [tableName: string]: { [key: string]: any}[];
  };
};
```

The fields are defined as follows:

BatchPutItem fields

BatchPutItem fields list

`operation`

The DynamoDB operation to perform. To perform the `BatchPutItem` DynamoDB operation, this must be set to `BatchPutItem`. This value is required.

`tables`

The DynamoDB tables to put the items in. Each table entry represents a list of DynamoDB items to insert for this specific table. At least one table must be provided. This value is required.

Things to remember:

- The fully inserted items are returned in the response, if successful.
- If an item hasn't been inserted in the table, a *null* element is displayed in the data block for that table.
- The inserted items are sorted per table, based on the order in which they were provided inside the request object.
- Each `Put` command inside a `BatchPutItem` is atomic, however, a batch can be partially processed. If a batch is partially processed due to an error, the unprocessed keys are returned as part of the invocation result inside the *unprocessedKeys* block.
- `BatchPutItem` is limited to 25 items.
- This operation **is not** supported when used with conflict detection. Using both at the same time may result in an error.

For the following example function request handler:

```
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 })],
    },
  };
}
```

The invocation result available in `ctx.result` is as follows:

```
{
   "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": []
   }
}
```

The `ctx.error` contains details about the error. The keys **data**, **unprocessedItems**, and each table key that was provided in the request object are guaranteed to be present in the invocation result.

Items that have been inserted are in the **data** block. Items that haven't been processed are marked as *null* inside the data block and are placed inside the **unprocessedItems** block.

TransactGetItems

The `TransactGetItems` request object lets you to tell the AWS AppSync DynamoDB function to make a `TransactGetItems` request to DynamoDB to retrieve multiple items, potentially across multiple tables. For this request object, you must specify the following:

- The table name of each request item where to retrieve the item from
- The key of each request item to retrieve from each table

The DynamoDB `TransactGetItems` limits apply and **no condition expression** can be provided.

The `TransactGetItems` request object has the following structure:

```
type DynamoDBTransactGetItemsRequest = {
  operation: 'TransactGetItems';
  transactItems: { table: string; key: { [key: string]: any }; projection?:
 { expression: string; expressionNames?: { [key: string]: string }; }[];
  };
};
```

The fields are defined as follows:

TransactGetItems fields

TransactGetItems fields list

`operation`

The DynamoDB operation to perform. To perform the `TransactGetItems` DynamoDB operation, this must be set to `TransactGetItems`. This value is required.

`transactItems`

The request items to include. The value is an array of request items. At least one request item must be provided. This `transactItems` value is required.

`table`

The DynamoDB table to retrieve the item from. The value is a string of the table name. This `table` value is required.

`key`

The DynamoDB key representing the primary key of the item to retrieve. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping).

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

Things to remember:

- If a transaction succeeds, the order of retrieved items in the `items` block will be the same as the order of request items.
- Transactions are performed in an all-or-nothing way. If any request item causes an error, the whole transaction will not be performed and error details will be returned.
- A request item being unable to be retrieved is not an error. Instead, a *null* element appears in the *items* block in the corresponding position.
- If the error of a transaction is *TransactionCanceledException*, the `cancellationReasons` block will be populated. The order of cancellation reasons in `cancellationReasons` block will be the same as the order of request items.
- `TransactGetItems` is limited to 100 request items.

For the following example function request handler:

```
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 }),
      },
    ],
  };
}
```

If the transaction succeeds and only the first requested item is retrieved, the invocation result available in `ctx.result` is as follows:

```
{
    "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
}
```

If the transaction fails due to *TransactionCanceledException* caused by the first request item, the invocation result available in `ctx.result` is as follows:

```
{
    "items": null,
    "cancellationReasons": [
       {
           "type":"Sample error type",
           "message":"Sample error message"
       },
       {
           "type":"None",
           "message":"None"
       }
```

```
    ]
}
```

The `ctx.error` contains details about the error. The keys **items** and **cancellationReasons** are guaranteed to be present in `ctx.result`.

TransactWriteItems

The `TransactWriteItems` request object lets you tell the AWS AppSync DynamoDB function to make a `TransactWriteItems` request to DynamoDB to write multiple items, potentially to multiple tables. For this request object, you must specify the following:

- The destination table name of each request item
- The operation of each request item to perform. There are four types of operations that are supported: *PutItem*, *UpdateItem*, *DeleteItem*, and *ConditionCheck*
- The key of each request item to write

The DynamoDB `TransactWriteItems` limits apply.

The `TransactWriteItems` request object has the following structure:

```
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 fields

TransactWriteItems fields list

The fields are defined as follows:

`operation`

The DynamoDB operation to perform. To perform the `TransactWriteItems` DynamoDB operation, this must be set to `TransactWriteItems`. This value is required.

`transactItems`

The request items to include. The value is an array of request items. At least one request item must be provided. This `transactItems` value is required.

For `PutItem`, the fields are defined as follows:

`table`

The destination DynamoDB table. The value is a string of the table name. This `table` value is required.

`operation`

The DynamoDB operation to perform. To perform the `PutItem` DynamoDB operation, this must be set to `PutItem`. This value is required.

`key`

The DynamoDB key representing the primary key of the item to put. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping). This value is required.

`attributeValues`

The rest of the attributes of the item to be put into DynamoDB. For more information about how to specify a “typed value”, see Type system (request mapping). This field is optional.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `PutItem` request overwrites any existing entry for that item. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see Transaction condition expressions. This value is optional.

For `UpdateItem`, the fields are defined as follows:

`table`

The DynamoDB table to update. The value is a string of the table name. This `table` value is required.

`operation`

The DynamoDB operation to perform. To perform the `UpdateItem` DynamoDB operation, this must be set to `UpdateItem`. This value is required.

`key`

The DynamoDB key representing the primary key of the item to update. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping). This value is required.

`update`

The `update` section lets you specify an update expression that describes how to update the item in DynamoDB. For more information about how to write update expressions, see the DynamoDB UpdateExpressions documentation. This section is required.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `UpdateItem` request updates the existing entry regardless of its current state. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see Transaction condition expressions. This value is optional.

For `DeleteItem`, the fields are defined as follows:

`table`

The DynamoDB table in which to delete the item. The value is a string of the table name. This `table` value is required.

`operation`

The DynamoDB operation to perform. To perform the `DeleteItem` DynamoDB operation, this must be set to `DeleteItem`. This value is required.

`key`

The DynamoDB key representing the primary key of the item to delete. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping). This value is required.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `DeleteItem` request deletes an item regardless of its current state. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see Transaction condition expressions. This value is optional.

For `ConditionCheck`, the fields are defined as follows:

`table`

The DynamoDB table in which to check the condition. The value is a string of the table name. This `table` value is required.

`operation`

The DynamoDB operation to perform. To perform the `ConditionCheck` DynamoDB operation, this must be set to `ConditionCheck`. This value is required.

`key`

The DynamoDB key representing the primary key of the item to condition check. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a "typed value", see Type system (request mapping). This value is required.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see Transaction condition expressions. This value is required.

Things to remember:

- Only keys of request items are returned in the response, if successful. The order of keys will be the same as the order of request items.
- Transactions are performed in an all-or-nothing way. If any request item causes an error, the whole transaction will not be performed and error details will be returned.
- No two request items can target the same item. Otherwise they will cause *TransactionCanceledException* error.
- If the error of a transaction is *TransactionCanceledException*, the `cancellationReasons` block will be populated. If a request item's condition check fails **and** you did not specify `returnValuesOnConditionCheckFailure` to be `false`, the item existing in the table will be retrieved and stored in `item` at the corresponding position of `cancellationReasons` block.
- `TransactWriteItems` is limited to 100 request items.
- This operation **is not** supported when used with conflict detection. Using both at the same time may result in an error.

For the following example function request handler:

```
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 }),
        },
      },
    ],
  };
}
```

If the transaction succeeds, the invocation result available in `ctx.result` is as follows:

```
{
    "keys": [
       // Key of the PutItem request
       {
           "post_id": "p1",
       },
       // Key of the UpdateItem request
       {
           "author_id": "a1"
       }
```

```
    ],
    "cancellationReasons": null
}
```

If the transaction fails due to condition check failure of the `PutItem` request, the invocation result available in `ctx.result` is as follows:

```
{
    "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"
       }
    ]
}
```

The `ctx.error` contains details about the error. The keys **keys** and **cancellationReasons** are guaranteed to be present in `ctx.result`.

Type system (request mapping)

When using the AWS AppSync DynamoDB function to call your DynamoDB tables, AWS AppSync needs to know the type of each value to use in that call. This is because DynamoDB supports more type primitives than GraphQL or JSON (such as sets and binary data). AWS AppSync needs some hints when translating between GraphQL and DynamoDB, otherwise it would have to make some assumptions on how data is structured in your table.

For more information about DynamoDB data types, see the DynamoDB Data type descriptors and Data types documentation.

A DynamoDB value is represented by a JSON object containing a single key-value pair. The key specifies the DynamoDB type, and the value specifies the value itself. In the following example, the key S denotes that the value is a string, and the value `identifier` is the string value itself.

```
{ "S" : "identifier" }
```

Note that the JSON object cannot have more than one key-value pair. If more than one key-value pair is specified, the request object isn't parsed.

A DynamoDB value is used anywhere in a request object where you need to specify a value. Some places where you need to do this include: `key` and `attributeValue` sections, and the `expressionValues` section of expression sections. In the following example, the DynamoDB String value `identifier` is being assigned to the `id` field in a `key` section (perhaps in a `GetItem` request object).

```
"key" : {
   "id" : { "S" : "identifier" }
}
```

Supported Types

AWS AppSync supports the following DynamoDB scalar, document, and set types:

String type S

A single string value. A DynamoDB String value is denoted by:

```
{ "S" : "some string" }
```

An example usage is:

```
"key" : {
   "id" : { "S" : "some string" }
}
```

String set type SS

A set of string values. A DynamoDB String Set value is denoted by:

```
{ "SS" : [ "first value", "second value", ... ] }
```

An example usage is:

```
"attributeValues" : {
   "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
}
```

Number type N

A single numeric value. A DynamoDB Number value is denoted by:

```
{ "N" : 1234 }
```

An example usage is:

```
"expressionValues" : {
   ":expectedVersion" : { "N" : 1 }
}
```

Number set type NS

A set of number values. A DynamoDB Number Set value is denoted by:

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

An example usage is:

```
"attributeValues" : {
   "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }
}
```

Binary type B

A binary value. A DynamoDB Binary value is denoted by:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

Note that the value is actually a string, where the string is the base64-encoded representation of the binary data. AWS AppSync decodes this string back into its binary value before sending it to DynamoDB. AWS AppSync uses the base64 decoding scheme as defined by RFC 2045: any character that isn't in the base64 alphabet is ignored.

An example usage is:

```
"attributeValues" : {
   "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

Binary set type BS

A set of binary values. A DynamoDB Binary Set value is denoted by:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

Note that the value is actually a string, where the string is the base64-encoded representation of the binary data. AWS AppSync decodes this string back into its binary value before sending it to DynamoDB. AWS AppSync uses the base64 decoding scheme as defined by RFC 2045: any character that is not in the base64 alphabet is ignored.

An example usage is:

```
"attributeValues" : {
   "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }
}
```

Boolean type BOOL

A Boolean value. A DynamoDB Boolean value is denoted by:

```
{ "BOOL" : true }
```

Note that only `true` and `false` are valid values.

An example usage is:

```
"attributeValues" : {
```

```
   "orderComplete" : { "BOOL" : false }
}
```

List type L

A list of any other supported DynamoDB value. A DynamoDB List value is denoted by:

```
{ "L" : [ ... ] }
```

Note that the value is a compound value, where the list can contain zero or more of any supported DynamoDB value (including other lists). The list can also contain a mix of different types.

An example usage is:

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

Map type M

Representing an unordered collection of key-value pairs of other supported DynamoDB values. A DynamoDB Map value is denoted by:

```
{ "M" : { ... } }
```

Note that a map can contain zero or more key-value pairs. The key must be a string, and the value can be any supported DynamoDB value (including other maps). The map can also contain a mix of different types.

An example usage is:

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
```

```
    }
}
```

Null type `NULL`

A null value. A DynamoDB Null value is denoted by:

```
{ "NULL" : null }
```

An example usage is:

```
"attributeValues" : {
   "phoneNumbers" : { "NULL" : null }
}
```

For more information about each type, see the DynamoDB documentation .

Type system (response mapping)

When receiving a response from DynamoDB, AWS AppSync automatically converts it into GraphQL and JSON primitive types. Each attribute in DynamoDB is decoded and returned in the response handler's context.

For example, if DynamoDB returns the following:

```
{
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

When the result is returned from your pipeline resolver, AWS AppSync converts it into GraphQL and JSON types as:

```
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

This section explains how AWS AppSync converts the following DynamoDB scalar, document, and set types:

String type S

A single string value. A DynamoDB String value is returned as a string.

For example, if DynamoDB returned the following DynamoDB String value:

```
{ "S" : "some string" }
```

AWS AppSync converts it to a string:

```
"some string"
```

String set type SS

A set of string values. A DynamoDB String Set value is returned as a list of strings.

For example, if DynamoDB returned the following DynamoDB String Set value:

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync converts it to a list of strings:

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

Number type N

A single numeric value. A DynamoDB Number value is returned as a number.

For example, if DynamoDB returned the following DynamoDB Number value:

```
{ "N" : 1234 }
```

AWS AppSync converts it to a number:

```
1234
```

Number set type NS

A set of number values. A DynamoDB Number Set value is returned as a list of numbers.

For example, if DynamoDB returned the following DynamoDB Number Set value:

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync converts it to a list of numbers:

```
[ 67.8, 12.2, 70 ]
```

Binary type B

A binary value. A DynamoDB Binary value is returned as a string containing the base64 representation of that value.

For example, if DynamoDB returned the following DynamoDB Binary value:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync converts it to a string containing the base64 representation of the value:

```
"SGVsbG8sIFdvcmxkIQo="
```

Note that the binary data is encoded in the base64 encoding scheme as specified in RFC 4648 and RFC 2045.

Binary set type BS

A set of binary values. A DynamoDB Binary Set value is returned as a list of strings containing the base64 representation of the values.

For example, if DynamoDB returned the following DynamoDB Binary Set value:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync converts it to a list of strings containing the base64 representation of the values:

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

Note that the binary data is encoded in the base64 encoding scheme as specified in RFC 4648 and RFC 2045.

Boolean type BOOL

A Boolean value. A DynamoDB Boolean value is returned as a Boolean.

For example, if DynamoDB returned the following DynamoDB Boolean value:

```
{ "BOOL" : true }
```

AWS AppSync converts it to a Boolean:

```
true
```

List type L

A list of any other supported DynamoDB value. A DynamoDB List value is returned as a list of values, where each inner value is also converted.

For example, if DynamoDB returned the following DynamoDB List value:

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

AWS AppSync converts it to a list of converted values:

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

Map type M

A key/value collection of any other supported DynamoDB value. A DynamoDB Map value is returned as a JSON object, where each key/value is also converted.

For example, if DynamoDB returned the following DynamoDB Map value:

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

AWS AppSync converts it to a JSON object:

```
{
   "someString" : "A string value",
   "someNumber" : 1,
   "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

Null type `NULL`

A null value.

For example, if DynamoDB returned the following DynamoDB Null value:

```
{ "NULL" : null }
```

AWS AppSync converts it to a null:

```
null
```

Filters

When querying objects in DynamoDB using the `Query` and `Scan` operations, you can optionally specify a `filter` that evaluates the results and returns only the desired values.

The filter property of a `Query` or `Scan` request has the following structure:

```
type DynamoDBExpression = {
```

```
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
};
```

The fields are defined as follows:

`expression`

The query expression. For more information about how to write filter expressions, see the DynamoDB QueryFilter and DynamoDB ScanFilter documentation. This field must be specified.

`expressionNames`

The substitutions for expression attribute *name* placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the `expression`. The value must be a string that corresponds to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the `expression`.

`expressionValues`

The substitutions for expression attribute *value* placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the `expression`, and the value must be a typed value. For more information about how to specify a "typed value", see Type system (request mapping). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the `expression`.

Example

The following example is a filter section for a request, where entries retrieved from DynamoDB are only returned if the title starts with the `title` argument.

Here we use the `util.transform.toDynamoDBFilterExpression` to automatically create a filter from an object:

```
const filter = util.transform.toDynamoDBFilterExpression({
  title: { beginsWith: 'far away' },
});

const request = {};
```

```
request.filter = JSON.parse(filter);
```

This generates the following filter:

```
{
  "filter": {
    "expression": "(begins_with(#title,:title_beginsWith))",
    "expressionNames": { "#title": "title" },
    "expressionValues": {
      ":title_beginsWith": { "S": "far away" }
    }
  }
}
```

Condition expressions

When you mutate objects in DynamoDB by using the `PutItem`, `UpdateItem`, and `DeleteItem` DynamoDB operations, you can optionally specify a condition expression that controls whether the request should succeed or not, based on the state of the object already in DynamoDB before the operation is performed.

The AWS AppSync DynamoDB function allows a condition expression to be specified in `PutItem`, `UpdateItem`, and `DeleteItem` request objects, and also a strategy to follow if the condition fails and the object was not updated.

Example 1

The following `PutItem` request object doesn't have a condition expression. As a result, it puts an item in DynamoDB even if an item with the same key already exists, thereby overwriting the existing item.

```
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),
  };
}
```

Example 2

The following `PutItem` object does have a condition expression that allows the operation succeed only if an item with the same key does *not* exist in DynamoDB.

```
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)" }
  };
}
```

By default, if the condition check fails, the AWS AppSync DynamoDB function provides an error for the mutation.

However, the AWS AppSync DynamoDB function offers some additional features to help developers handle some common edge cases:

- If AWS AppSync DynamoDB functions can determine that the current value in DynamoDB matches the desired result, it treats the operation as if it succeeded anyway.
- Instead of returning an error, you can configure the function to invoke a custom Lambda function to decide how the AWS AppSync DynamoDB function should handle the failure.

These are described in greater detail in the Handling a condition check failure section.

For more information about DynamoDB conditions expressions, see the DynamoDB ConditionExpressions documentation .

Specifying a condition

The `PutItem`, `UpdateItem`, and `DeleteItem` request objects all allow an optional `condition` section to be specified. If omitted, no condition check is made. If specified, the condition must be true for the operation to succeed.

A `condition` section has the following structure:

```
type ConditionCheckExpression = {
```

```
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
  equalsIgnore?: string[];
  consistentRead?: boolean;
  conditionalCheckFailedHandler?: {
    strategy: 'Custom' | 'Reject';
    lambdaArn?: string;
  };
};
```

The following fields specify the condition:

`expression`

The update expression itself. For more information about how to write condition expressions, see the DynamoDB ConditionExpressions documentation . This field must be specified.

`expressionNames`

The substitutions for expression attribute name placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the *expression*, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the *expression*.

`expressionValues`

The substitutions for expression attribute value placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the expression, and the value must be a typed value. For more information about how to specify a “typed value”, see Type system (request mapping). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the expression.

The remaining fields tell the AWS AppSync DynamoDB function how to handle a condition check failure:

`equalsIgnore`

When a condition check fails when using the `PutItem` operation, the AWS AppSync DynamoDB function compares the item currently in DynamoDB against the item it tried to write. If they are the same, it treats the operation as it if succeeded anyway. You can use the `equalsIgnore`

field to specify a list of attributes that AWS AppSync should ignore when performing that comparison. For example, if the only difference was a `version` attribute, it treats the operation as if it succeeded. This field is optional.

`consistentRead`

When a condition check fails, AWS AppSync gets the current value of the item from DynamoDB using a strongly consistent read. You can use this field to tell the AWS AppSync DynamoDB function to use an eventually consistent read instead. This field is optional, and defaults to `true`.

`conditionalCheckFailedHandler`

This section allows you to specify how the AWS AppSync DynamoDB function treats a condition check failure after it has compared the current value in DynamoDB against the expected result. This section is optional. If omitted, it defaults to a strategy of `Reject`.

`strategy`

The strategy the AWS AppSync DynamoDB function takes after it has compared the current value in DynamoDB against the expected result. This field is required and has the following possible values:

`Reject`

The mutation fails, and an error is added to the GraphQL response.

`Custom`

The AWS AppSync DynamoDB function invokes a custom Lambda function to decide how to handle the condition check failure. When the `strategy` is set to `Custom`, the `lambdaArn` field must contain the ARN of the Lambda function to invoke.

`lambdaArn`

The ARN of the Lambda function to invoke that determines how the AWS AppSync DynamoDB function should handle the condition check failure. This field must only be specified when `strategy` is set to `Custom`. For more information about how to use this feature, see Handling a condition check failure.

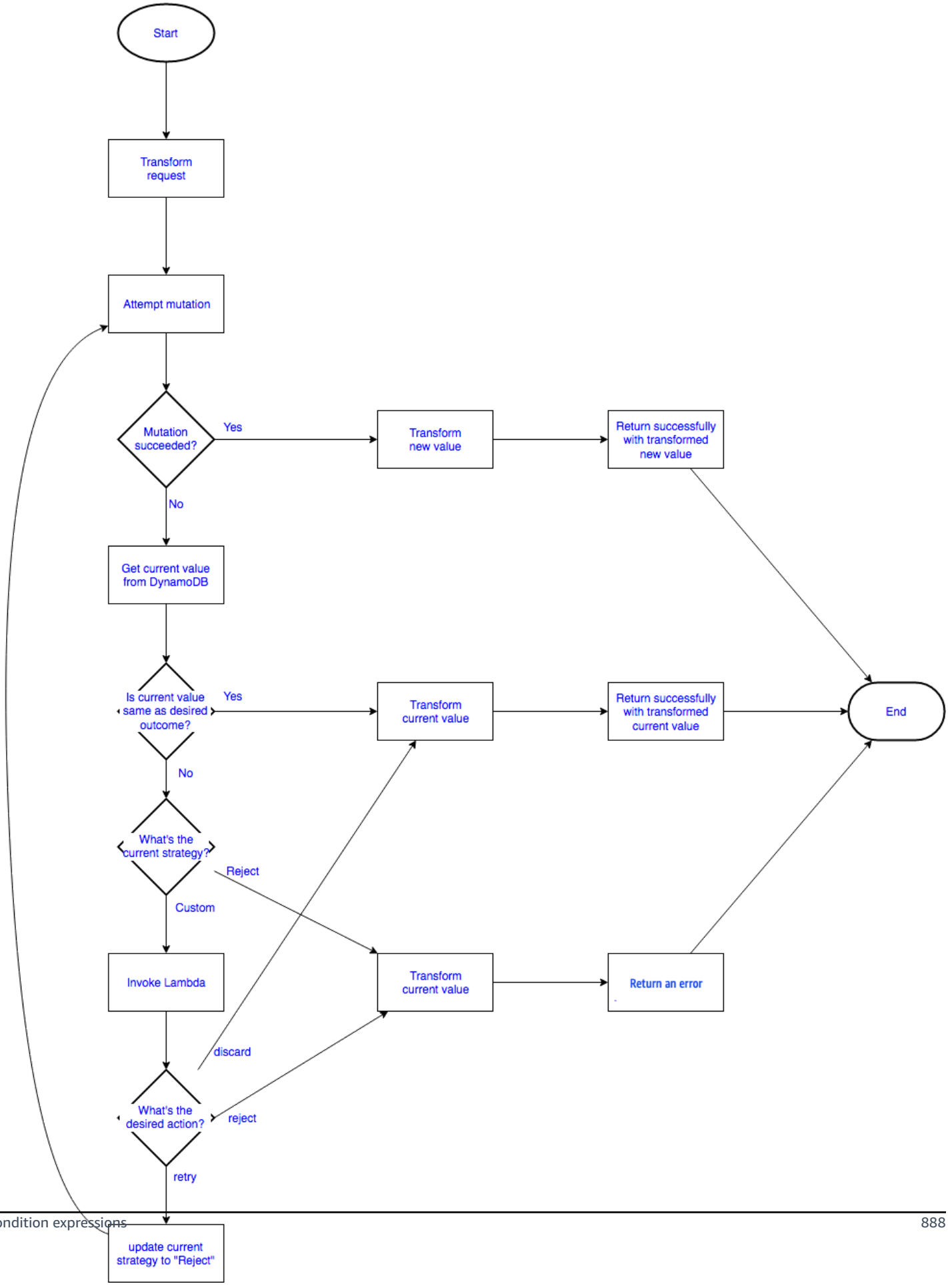
Handling a condition check failure

When a condition check fails, the AWS AppSync DynamoDB function can pass on the error for the mutation and the current value of the object by using the `util.appendError` utility. However,

the AWS AppSync DynamoDB function offers some additional features to help developers handle some common edge cases:

- If AWS AppSync DynamoDB functions can determine that the current value in DynamoDB matches the desired result, it treats the operation as if it succeeded anyway.
- Instead of returning an error, you can configure the function to invoke a custom Lambda function to decide how the AWS AppSync DynamoDB function should handle the failure.

The flowchart for this process is:


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"

Checking for the desired result

When the condition check fails, the AWS AppSync DynamoDB function performs a `GetItem` DynamoDB request to get the current value of the item from DynamoDB. By default, it uses a strongly consistent read, however this can be configured using the `consistentRead` field in the `condition` block and compare it against the expected result:

- For the `PutItem` operation, the AWS AppSync DynamoDB function compares the current value against the one it attempted to write, excluding any attributes listed in `equalsIgnore` from the comparison. If the items are the same, it treats the operation as successful and returns the item that was retrieved from DynamoDB. Otherwise, it follows the configured strategy.

  For example, if the `PutItem` request object looked like the following:

```
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']
    }
  };
}
```

  And the item currently in DynamoDB looked like the following:

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

  The AWS AppSync DynamoDB function would compare the item it tried to write against the current value, see that the only difference was the `version` field, but because it's configured

to ignore the `version` field, it treats the operation as successful and returns the item that was retrieved from DynamoDB.

- For the `DeleteItem` operation, the AWS AppSync DynamoDB function checks to verify that an item was returned from DynamoDB. If no item was returned, it treats the operation as successful. Otherwise, it follows the configured strategy.
- For the `UpdateItem` operation, the AWS AppSync DynamoDB function does not have enough information to determine if the item currently in DynamoDB matches the expected result, and therefore follows the configured strategy.

If the current state of the object in DynamoDB is different from the expected result, the AWS AppSync DynamoDB function follows the configured strategy, to either reject the mutation or invoke a Lambda function to determine what to do next.

Following the “reject” strategy

When following the `Reject` strategy, the AWS AppSync DynamoDB function returns an error for the mutation.

For example, given the following mutation request:

```
mutation {
    updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
        Name
        theVersion
    }
}
```

If the item returned from DynamoDB looks like the following:

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

And the function response handler looks like the following:

```
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
}
```

The GraphQL response looks like the following:

```
{
  "data": null,
  "errors": [
    {
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      ...
    }
  ]
}
```

Also, if any fields in the returned object are filled by other resolvers and the mutation had succeeded, they won't be resolved when the object is returned in the `error` section.

Following the "custom" strategy

When following the `Custom` strategy, the AWS AppSync DynamoDB function invokes a Lambda function to decide what to do next. The Lambda function chooses one of the following options:

- `reject` the mutation. This tells the AWS AppSync DynamoDB function to behave as if the configured strategy was `Reject`, returning an error for the mutation and the current value of the object in DynamoDB as described in the previous section.
- `discard` the mutation. This tells the AWS AppSync DynamoDB function to silently ignore the condition check failure and returns the value in DynamoDB.
- `retry` the mutation. This tells the AWS AppSync DynamoDB function to retry the mutation with a new request object.

The Lambda invocation request

The AWS AppSync DynamoDB function invokes the Lambda function specified in the `lambdaArn`. It uses the same `service-role-arn` configured on the data source. The payload of the invocation has the following structure:

```
{
    "arguments": { ... },
    "requestMapping": {... },
    "currentValue": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

The fields are defined as follows:

`arguments`

The arguments from the GraphQL mutation. This is the same as the arguments available to the request object in `context.arguments`.

`requestMapping`

The request object for this operation.

`currentValue`

The current value of the object in DynamoDB.

`resolver`

Information about the AWS AppSync resolver or function.

`identity`

Information about the caller. This is the same as the identity information available to the request object in `context.identity`.

A full example of the payload:

```
{
    "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": "AIDAAAAAAAAAAAAAAAAAA",
        "userArn": "arn:aws:iam::123456789012:user/appsync"
    }
}
```

The Lambda Invocation Response

The Lambda function can inspect the invocation payload and apply any business logic to decide how the AWS AppSync DynamoDB function should handle the failure. There are three options for handling the condition check failure:

- `reject` the mutation. The response payload for this option must have this structure:

  ```
  {
      "action": "reject"
  }
  ```

  This tells the AWS AppSync DynamoDB function to behave as if the configured strategy was `Reject`, returning an error for the mutation and the current value of the object in DynamoDB, as described in the section above.
- `discard` the mutation. The response payload for this option must have this structure:

  ```
  {
      "action": "discard"
  }
  ```

  This tells the AWS AppSync DynamoDB function to silently ignore the condition check failure and returns the value in DynamoDB.
- `retry` the mutation. The response payload for this option must have this structure:

  ```
  {
      "action": "retry",
      "retryMapping": { ... }
  }
  ```

  This tells the AWS AppSync DynamoDB function to retry the mutation with a new request object. The structure of the `retryMapping` section depends on the DynamoDB operation, and is a subset of the full request object for that operation.

  For `PutItem`, the `retryMapping` section has the following structure. For a description of the `attributeValues` field, see PutItem.

  ```
  {
      "attributeValues": { ... },
      "condition": {
          "equalsIgnore" = [ ... ],
  ```

```
        "consistentRead" = true
    }
}
```

For `UpdateItem`, the `retryMapping` section has the following structure. For a description of the `update` section, see UpdateItem.

```
{
    "update" : {
        "expression" : "someExpression"
        "expressionNames" : {
            "#foo" : "foo"
        },
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "condition": {
        "consistentRead" = true
    }
}
```

For `DeleteItem`, the `retryMapping` section has the following structure.

```
{
    "condition": {
        "consistentRead" = true
    }
}
```

There is no way to specify a different operation or key to work on. The AWS AppSync DynamoDB function only allows retries of the same operation on the same object. Also, the `condition` section doesn't allow a `conditionalCheckFailedHandler` to be specified. If the retry fails, the AWS AppSync DynamoDB function follows the `Reject` strategy.

Here is an example Lambda function to deal with a failed `PutItem` request. The business logic looks at who made the call. If it was made by `jeffTheAdmin`, it retries the request, updating the `version` and `expectedVersion` from the item currently in DynamoDB. Otherwise, it rejects the mutation.

```
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)
};
```

Transaction condition expressions

Transaction condition expressions are available in requests of all four types of operations in `TransactWriteItems`, namely, `PutItem`, `DeleteItem`, `UpdateItem`, and `ConditionCheck`.

For `PutItem`, `DeleteItem`, and `UpdateItem`, the transaction condition expression is optional. For `ConditionCheck`, the transaction condition expression is required.

Example 1

The following transactional `DeleteItem` function request handler does not have a condition expression. As a result, it deletes the item in DynamoDB.

```
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 }),
      }
    ],
  };
}
```

Example 2

The following transactional `DeleteItem` function request handler does have a transaction condition expression that allows the operation succeed only if the author of that post equals a certain name.

```
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 },
        }),
      }
```

```
    ],
  };
}
```

If the condition check fails, it will cause `TransactionCanceledException` and the error detail will be returned in `ctx.result.cancellationReasons`. Note that by default, the old item in DynamoDB that made condition check fail will be returned in `ctx.result.cancellationReasons`.

Specifying a condition

The `PutItem`, `UpdateItem`, and `DeleteItem` request objects all allow an optional `condition` section to be specified. If omitted, no condition check is made. If specified, the condition must be true for the operation to succeed. The `ConditionCheck` must have a `condition` section to be specified. The condition must be true for the whole transaction to succeed.

A `condition` section has the following structure:

```
type TransactConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string };
  expressionValues?: { [key: string]: string };
  returnValuesOnConditionCheckFailure: boolean;
};
```

The following fields specify the condition:

`expression`

The update expression itself. For more information about how to write condition expressions, see the DynamoDB ConditionExpressions documentation . This field must be specified.

`expressionNames`

The substitutions for expression attribute name placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the *expression*, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the *expression*.

`expressionValues`

The substitutions for expression attribute value placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the expression, and the value must be a typed value. For more information about how to specify a "typed value", see Type system (request mapping). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the expression.

`returnValuesOnConditionCheckFailure`

Specify whether to retrieve the item in DynamoDB back when a condition check fails. The retrieved item will be in `ctx.result.cancellationReasons[<index>].item`, where `<index>` is the index of the request item that failed the condition check. This value defaults to true.

Projections

When reading objects in DynamoDB using the `GetItem`, `Scan`, `Query`, `BatchGetItem`, and `TransactGetItems` operations, you can optionally specify a projection that identifies the attributes that you want. The projection property has the following structure, which is similar to filters:

```
type DynamoDBExpression = {
  expression: string;
  expressionNames?: { [key: string]: string}
};
```

The fields are defined as follows:

`expression`

The projection expression, which is a string. To retrieve a single attribute, specify its name. For multiple attributes, the names must be comma-separated values. For more information on writing projection expressions, see the DynamoDB projection expressions documentation. This field is required.

`expressionNames`

The substitutions for expression attribute *name* placeholders in the form of key-value pairs. The key corresponds to a name placeholder used in the `expression`. The value must be a string that corresponds to the attribute name of the item in DynamoDB. This field is optional

and should only be populated with substitutions for expression attribute name placeholders used in the `expression`. For more information about `expressionNames`, see the DynamoDB documentation.

Example 1

The following example is a projection section for a JavaScript function in which only the attributes `author` and `id` are returned from DynamoDB:

```
projection : {
    expression : "#author, id",
    expressionNames : {
        "#author" : "author"
    }
}
```

Tip

You can access your GraphQL request selection set using selectionSetList. This field allows you to frame your projection expression dynamically according to your requirements.

Note

While using projection expressions with the `Query` and `Scan` operations, the value for `select` must be `SPECIFIC_ATTRIBUTES`. For more information, see the DynamoDB documentation.

AWS AppSync JavaScript resolver function reference for OpenSearch

The AWS AppSync resolver for Amazon OpenSearch Service enables you to use GraphQL to store and retrieve data in existing OpenSearch Service domains in your account. This resolver works by allowing you to map an incoming GraphQL request into an OpenSearch Service request, and then map the OpenSearch Service response back to GraphQL. This section describes the function request and response handlers for the supported OpenSearch Service operations.

Request

Most OpenSearch Service request objects have a common structure where just a few pieces change. The following example runs a search against an OpenSearch Service domain, where documents are of type `post` and are indexed under `id`. The search parameters are defined in the `body` section, with many of the common query clauses being defined in the `query` field. This example will search for documents containing `"Nadia"`, or `"Bailey"`, or both, in the `author` field of a document:

```
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' } },
            ],
          },
        },
      },
    },
  };
}
```

Response

As with other data sources, OpenSearch Service sends a response to AWS AppSync that needs to be converted to GraphQL. .

Most GraphQL queries are looking for the `_source` field from an OpenSearch Service response. Because you can do searches to return either an individual document or a list of documents, there are two common response patterns used in OpenSearch Service:

List of Results

```
export function response(ctx) {
  const entries = [];
  for (const entry of ctx.result.hits.hits) {
    entries.push(entry['_source']);
  }
  return entries;
}
```

Individual Item

```
export function response(ctx) {
  return ctx.result['_source']
}
```

`operation` field

> **Note**
>
> This applies only to the Request handler.

HTTP method or verb (GET, POST, PUT, HEAD or DELETE) that AWS AppSync sends to the OpenSearch Service domain. Both the key and the value must be a string.

```
"operation" : "PUT"
```

`path` field

> **Note**
>
> This applies only to the Request handler.

The search path for an OpenSearch Service request from AWS AppSync. This forms a URL for the operation's HTTP verb. Both the key and the value must be strings.

```
"path" : "/indexname/type"
```

```
"path" : "/indexname/type/_search"
```

When the request handler is evaluated, this path is sent as part of the HTTP request, including the OpenSearch Service domain. For example, the previous example might translate to:

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

`params` field

> **Note**
>
> This applies only to the Request handler.

Used to specify what action your search performs, most commonly by setting the **query** value inside of the **body**. However, there are several other capabilities that can be configured, such as the formatting of responses.

- **headers**

  The header information, as key-value pairs. Both the key and the value must be strings. For example:

  ```
  "headers" : {
      "Content-Type" : "application/json"
  }
  ```

  > **Note**
  >
  > AWS AppSync currently supports only JSON as a `Content-Type`.

- **queryString**

  Key-value pairs that specify common options, such as code formatting for JSON responses. Both the key and the value must be a string. For example, if you want to get pretty-formatted JSON, you would use:

  ```
  "queryString" : {
  ```

```
    "pretty" : "true"
}
```

- **body**

  This is the main part of your request, allowing AWS AppSync to craft a well-formed search request to your OpenSearch Service domain. The key must be a string comprised of an object. A couple of demonstrations are shown below.

Example 1

Return all documents with a city matching “seattle”:

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: { from: 0, size: 50, query: { match: { city: 'seattle' } } },
    },
  };
}
```

Example 2

Return all documents matching “washington” as the city or the state:

```
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'] },
        },
```

```
      },
    },
  };
}
```

Passing variables

> **Note**
>
> This applies only to the Request handler.

You can also pass variables as part of evaluation in your request handler. For example, suppose you had a GraphQL query such as the following:

```
query {
    searchForState(state: "washington"){
        ...
    }
}
```

The function request handler could be the following:

```
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'] },
        },
      },
    },
  };
}
```

AWS AppSync JavaScript resolver function reference for Lambda

You can use AWS AppSync functions and resolvers to invoke Lambda functions located in your account. You can shape your request payloads and the response from your Lambda functions before returning them to your clients. You can also specify the type of operation to perform in your request object. This section describes the requests for the supported Lambda operations.

Request object

The Lambda request object handles fields related to your Lambda function:

```
export type LambdaRequest = {
  operation: 'Invoke' | 'BatchInvoke';
  invocationType?: 'RequestResponse' | 'Event';
  payload: unknown;
};
```

Here's an example that uses an `invoke` operation with its payload data being the `getPost` field from a GraphQL schema along with its arguments from the context:

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

The entire mapping document is passed as the input to your Lambda function so that the previous example now looks like this:

```
{
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "input": {
        "id": "postId1",
      }
    }
```

```
  }
}
```

Operation

The Lambda data source lets you define two operations in the `operation` field: `Invoke` and `BatchInvoke`. The `Invoke` operation lets AWS AppSync know to call your Lambda function for every GraphQL field resolver. `BatchInvoke` instructs AWS AppSync to batch requests for the current GraphQL field. The `operation` field is required.

For `Invoke`, the resolved request matches the input payload of the Lambda function. Let's modify the example above:

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

This is resolved and passed to the Lambda function, which could look something like this:

```
{
  "operation": "Invoke",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

For `BatchInvoke`, the request is applied to every field resolver in the batch. For conciseness, AWS AppSync merges all the request `payload` values into a list under a single object matching the request object. The following example request handler shows the merge:

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: ctx,
  };
}
```

This request is evaluated and resolved into the following mapping document:

```
{
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

Each element of the `payload` list corresponds to a single batch item. The Lambda function is also expected to return a list-shaped response matching the order of the items sent in the request:

```
[
  { "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
]
```

Payload

The `payload` field is a container used to pass any data to the Lambda function. If the `operation` field is set to `BatchInvoke`, AWS AppSync wraps the existing `payload` values into a list. The `payload` field is optional.

Invocation type

The Lambda data source allows you to define two invocation types: `RequestResponse` and `Event`. The invocation types are synonymous with the invocation types defined in the Lambda API. The `RequestResponse` invocation type lets AWS AppSync call your Lambda function synchronously to wait for a response. The `Event` invocation allows you to invoke your Lambda function asynchronously. For more information on how Lambda handles `Event` invocation type requests, see Asynchronous invocation. The `invocationType` field is optional. If this field is not included in the request, AWS AppSync will default to the `RequestResponse` invocation type.

For any `invocationType` field, the resolved request matches the input payload of the Lambda function. Let's modify the example above:

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    invocationType: 'Event',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

This is resolved and passed to the Lambda function, which could look something like this:

```
{
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

When the `BatchInvoke` operation is used in conjunction with the `Event` invocation type field, AWS AppSync merges the field resolver in the same way mentioned above, and the request is passed to your Lambda function as an asynchronous event with the `payload` being a list of values. The response from an `Event` invocation type request results in a `null` value without a response handler:

```
{
  "data": {
    "field": null
  }
}
```

We recommend that you disable resolver caching for `Event` invocation type resolvers because these would not be sent to Lambda if there were a cache hit.

Response object

As with other data sources, your Lambda function sends a response to AWS AppSync that must be converted to a GraphQL type. The result of the Lambda function is contained in the `context` result property (`context.result`).

If the shape of your Lambda function response matches the shape of the GraphQL type, you can forward the response using the following function response handler:

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

There are no required fields or shape restrictions that apply to the response object. However, because GraphQL is strongly typed, the resolved response must match the expected GraphQL type.

Lambda function batched response

If the `operation` field is set to `BatchInvoke`, AWS AppSync expects a list of items back from the Lambda function. In order for AWS AppSync to map each result back to the original request item, the response list must match in size and order. It's valid to have `null` items in the response list; `ctx.result` is set to *null* accordingly.

AWS AppSync JavaScript resolver function reference for EventBridge data source

The AWS AppSync resolver function request and response used with the EventBridge data source allows you to send custom events to the Amazon EventBridge bus.

Request

The request handler allows you to send multiple custom events to an EventBridge event bus:

```
export function request(ctx) {
  return {
    "operation" : "PutEvents",
    "events" : [{}]
  }
}
```

An EventBridge `PutEvents` request has the following type definition:

```
type PutEventsRequest = {
  operation: 'PutEvents'
```

```
  events: {
    source: string
    detail: { [key: string]: any }
    detailType: string
    resources?: string[]
    time?: string // RFC3339 Timestamp format
  }[]
}
```

Response

If the `PutEvents` operation is successful, the response from EventBridge is included in the `ctx.result`:

```
export function response(ctx) {
  if(ctx.error)
    util.error(ctx.error.message, ctx.error.type, ctx.result)
  else
    return ctx.result
}
```

Errors that occur while performing `PutEvents` operations such as `InternalExceptions` or `Timeouts` will appear in `ctx.error`. For a list of EventBridge's common errors, see the EventBridge common error reference.

The `result` will have the following type definition:

```
type PutEventsResult = {
  Entries: {
    ErrorCode: string
    ErrorMessage: string
    EventId: string
  }[]
  FailedEntryCount: number
}
```

- **Entries**

  The ingested event results, both successful and unsuccessful. If the ingestion was successful, the entry has the `EventID` in it. Otherwise, you can use the `ErrorCode` and `ErrorMessage` to identify the problem with the entry.

For each record, the index of the response element is the same as the index in the request array.

- **FailedEntryCount**

  The number of failed entries. This value is represented as an integer.

For more information about the response of `PutEvents`, see PutEvents.

Example sample response 1

The following example is a `PutEvents` operation with two successful events:

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ],
    "FailedEntryCount" : 0
}
```

Example sample response 2

The following example is a `PutEvents` operation with three events, two successes and one fail:

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        },
        {
            "ErrorCode" : "SampleErrorCode",
            "ErrorMessage" : "Sample Error Message"
        }
    ],
    "FailedEntryCount" : 1
```

```
}
```

PutEvents fields

PutEvents contains the following mapping template fields:

- **Version**

  Common to all request mapping templates, the version field defines the version that the template uses. This field is required. The value 2018-05-29 is the only version supported for the EventBridge mapping templates.
- **Operation**

  The only supported operation is PutEvents. This operation allows you to add custom events to your event bus.
- **Events**

  An array of events that will be added to the event bus. This array should have an allocation of 1 - 10 items.

  The Event object has the following fields:
  - "source": A string that defines the source of the event.
  - "detail": A JSON object that you can use to attach information about the event. This field can be an empty map ({ }).
  - "detailType: A string that identifies the type of event.
  - "resources": A JSON array of strings that identifies resources involved in the event. This field can be an empty array.
  - "time": The event timestamp provided as a string. This should follow the RFC3339 timestamp format.

The snippets below are some examples of valid Event objects:

Example 1

```
{
    "source" : "source1",
    "detail" : {
        "key1" : [1,2,3,4],
```

```
        "key2" : "strval"
    },
    "detailType" : "sampleDetailType",
    "resources" : ["Resouce1", "Resource2"],
    "time" : "2022-01-10T05:00:10Z"
}
```

Example 2

```
{
    "source" : "source1",
    "detail" : {},
    "detailType" : "sampleDetailType"
}
```

Example 3

```
{
    "source" : "source1",
    "detail" : {
        "key1" : 1200
    },
    "detailType" : "sampleDetailType",
    "resources" : []
}
```

AWS AppSync JavaScript resolver function reference for None data source

The AWS AppSync resolver function request and response with the data source of type *None* enables you to shape requests for AWS AppSync local operations.

Request

The request handler can be simple and enables you to pass as much contextual information as possible via the `payload` field.

```
type NONERequest = {
  payload: any;
```

```
};
```

Here is an example where the field arguments are passed to the payload:

```
export function request(ctx) {
  return {
    payload: context.args
  };
}
```

The value of the `payload` field will be forwarded to the function response handler and is available in `context.result`.

Payload

The `payload` field is a container that can be used to pass any data that is then made available to the function response handler.

The `payload` field is optional.

Response

Because there is no data source, the value of the `payload` field will be forwarded to the function response handler and set on the `context.result` property.

If the shape of the `payload` field value exactly matches the shape of the GraphQL type, you can forward the response using the following response handler:

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

There are no required fields or shape restrictions that apply to the return response. However, because GraphQL is strongly typed, the resolved response must match the expected GraphQL type.

AWS AppSync JavaScript resolver function reference for HTTP

The AWS AppSync HTTP resolver functions enable you to send requests from AWS AppSync to any HTTP endpoint, and responses from your HTTP endpoint back to AWS AppSync. With your request

handler, you can provide hints to AWS AppSync about the nature of the operation to be invoked. This section describes the different configurations for the supported HTTP resolver.

Request

```
type HTTPRequest = {
  method: 'PUT' | 'POST' | 'GET' | 'DELETE' | 'PATCH';
  params?: {
    query?: { [key: string]: any };
    headers?: { [key: string]: string };
    body?: any;
  };
  resourcePath: string;
};
```

The following snippet is an example of an HTTP POST request, with a `text/plain` body:

```
export function request(ctx) {
  return {
    method: 'POST',
    params: {
      headers: { 'Content-Type': 'text/plain' },
      body: 'this is an example of text body',
    },
    resourcePath: '/',
  };
}
```

Method

> **Note**
>
> This applies only to the Request handler.

HTTP method or verb (GET, POST, PUT, PATCH, or DELETE) that AWS AppSync sends to the HTTP endpoint.

```
"method": "PUT"
```

ResourcePath

> **Note**
>
> This applies only to the Request handler.

The resource path that you want to access. Along with the endpoint in the HTTP data source, the resource path forms the URL that the AWS AppSync service makes a request to.

```
"resourcePath": "/v1/users"
```

When the request is evaluated, this path is sent as part of the HTTP request, including the HTTP endpoint. For example, the previous example might translate to the following:

```
PUT <endpoint>/v1/users
```

Params fields

> **Note**
>
> This applies only to the Request handler.

Used to specify what action your search performs, most commonly by setting the **query** value inside the **body**. However, there are several other capabilities that can be configured, such as the formatting of responses.

headers

The header information, as key-value pairs. Both the key and the value must be strings.

For example:

```
"headers" : {
    "Content-Type" : "application/json"
}
```

Currently supported `Content-Type` headers are:

```
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
```

You can't set the following HTTP headers:

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

Key-value pairs that specify common options, such as code formatting for JSON responses. Both the key and the value must be a string. The following example shows how you can send a query string as `?type=json`:

```
"query" : {
    "type" : "json"
}
```

body

The body contains the HTTP request body that you choose to set. The request body is always a UTF-8 encoded string unless the content type specifies the charset.

```
"body":"body string"
```

Response

See an example here.

AWS AppSync JavaScript resolver function reference for Amazon RDS

The AWS AppSync RDS function and resolver allows developers to send SQL queries to an Amazon Aurora cluster database using the RDS Data API and get back the result of these queries. You can write SQL statements that are sent to the Data API by using AWS AppSync's `rds` module `sql`-tagged template or by using the `rds` module's `select`, `insert`, `update`, and `remove` helper functions. AWS AppSync utilizes the RDS Data Service's `ExecuteStatement` action to run SQL statements against the database.

Topics

SQL tagged template

AWS AppSync's `sql` tagged template enables you to create a static statement that can receive dynamic values at runtime by using template expressions. AWS AppSync builds a variable map from the expression values to construct a `SqlParameterized` query that is sent to the Amazon Aurora Serverless Data API. With this method, it isn't possible for dynamic values passed at run time to modify the original statement, which could cause unintented execution. All dynamic values are passed as parameters, can't modify the original statement, and aren't executed by the database. This makes your query less vulnerable to SQL injection attacks.

Note

In all cases, when writing SQL statements, you should follow security guidelines to properly handle data that you receive as input.

Note

The `sql` tagged template only supports passing variable values. You can't use an expression to dynamically specify the column or table names. However, you can use utility functions to build dynamic statements.

In the following example, we create a query that filters based on the value of the `col` argument that is set dynamically in the GraphQL query at run time. The value can only be added to the statement using the tag expression:

```
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);
}
```

By passing all dynamic values through the variable map, we rely on the database engine to securely handle and sanitize values.

Creating statements

Functions and resolvers can interact with MySQL and PostgreSQL databases. Use `createMySQLStatement` and `createPgStatement` respectively to build statements. For example, `createMySQLStatement` can create a MySQL query. These functions accept up to two statements, useful when a request should retrieve results immediately. With MySQL, you could do:

```
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` and `createMySQLStatement` does not escape or quote statements built with the `sql` tagged template.

Retrieving data

The result of your executed SQL statement is available in your response handler in the `context.result` object. The result is a JSON string with the response elements from the `ExecuteStatement` action. When parsed, the result has the following shape:

```
type SQLStatementResults = {
    sqlStatementResults: {
        records: any[];
        columnMetadata: any[];
        numberOfRecordsUpdated: number;
        generatedFields?: any[]
    }[]
}
```

You can use the `toJsonObject` utility to transform the result into a list of JSON objects representing the returned rows. For example:

```
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]
}
```

Note that `toJsonObject` returns an array of statement results. If you provided one statement, the array length is `1`. If you provided two statements, the array length is `2`. Each result in the array contains `0` or more rows. `toJsonObject` returns `null` if the result value is invalid or unexpected.

Utility functions

You can use the AWS AppSync RDS module's utility helpers to interact with your database.

SQL Select

The `select` utility creates a `SELECT` statement to query your relational database.

Basic use

In its basic form, you can specify the table you want to query:

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // "SELECT * FROM "persons"
    return createPgStatement(select({table: 'persons'}));
}
```

Note that you can also specify the schema in your table identifier:

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // SELECT * FROM "private"."persons"
    return createPgStatement(select({table: 'private.persons'}));
}
```

Specifying columns

You can specify columns with the `columns` property. If this isn't set to a value, it defaults to *:

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name']
    }));
}
```

You can specify a column's table as well:

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "persons"."name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'persons.name']
    }));
}
```

Limits and offsets

You can apply `limit` and `offset` to the query:

```
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

You can sort your results with the `orderBy` property. Provide an array of objects specifying the column and an optional `dir` property:

```
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'}]
    }));
}
```

Filters

You can build filters by using the special condition object:

```
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'}}
    }));
}
```

You can also combine filters:

```
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}}
    }));
}
```

You can also create OR statements:

```
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 } }
        ]}
    }));
}
```

You can also negate a condition with `not`:

```
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 } }
        ]}
    }));
```

```
}
```

You can also use the following operators to compare values:

Operator	Description	Possible value types
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

The `insert` utility provides a straightforward way of inserting single row items in your database with the `INSERT` operation.

Single item insertions

To insert an item, specify the table and then pass in your object of values. The object keys are mapped to your table columns. Columns names are automatically escaped, and values are sent to the database using the variable map:

```
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 use case

You can combine an `insert` followed by a `select` to retrieve your inserted row:

```
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 use case

With Postgres, you can use `returning` to obtain data from the row that you inserted. It accepts * or an array of column names:

```
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

The `update` utility allows you to update existing rows. You can use the condition object to apply changes to the specified columns in all the rows that satisfy the condition. For example, let's say we have a schema that allows us to make this mutation. We want to update the `name` of `Person` with the `id` value of `3` but only if we've known them (`known_since`) since the year `2000`:

```
mutation Update {
    updatePerson(
        input: {id: 3, name: "Jon"},
        condition: {known_since: {ge: "2000"}}
    ) {
    id
    name
  }
}
```

Our update resolver looks like this:

```
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)
}
```

We can add a check to our condition to make sure that only the row that has the primary key `id` equal to 3 is updated. Similarly, for Postgres `inserts`, you can use `returning` to return the modified data.

SQL Delete

The `remove` utility allows you to delete existing rows. You can use the condition object on all rows that satisfy the condition. Note that `delete` is a reserved keyword in JavaScript. `remove` should be used instead:

```
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)
```

```
}
```

Casting

In some cases, you may want more specificity about the correct object type to use in your statement. You can use the provided type hints to specify the type of your parameters. AWS AppSync supports the same type hints as the Data API. You can cast your parameters by using the `typeHint` functions from the AWS AppSync `rds` module.

The following example allows you to send an array as a value that is casted as a JSON object. We use the `->` operator to retrieve the element at the `index 2` in the JSON array:

```
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
}
```

Casting is also useful when handling and comparing `DATE`, `TIME`, and `TIMESTAMP`:

```
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)
}
```

Here's another example showing how you can send the current date and time:

```
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)}`)
}
```

Available type hints

- `typeHint.DATE` - The corresponding parameter is sent as an object of the `DATE` type to the database. The accepted format is `YYYY-MM-DD`.
- `typeHint.DECIMAL` - The corresponding parameter is sent as an object of the `DECIMAL` type to the database.
- `typeHint.JSON` - The corresponding parameter is sent as an object of the `JSON` type to the database.
- `typeHint.TIME` - The corresponding string parameter value is sent as an object of the `TIME` type to the database. The accepted format is `HH:MM:SS[.FFF]`.
- `typeHint.TIMESTAMP` - The corresponding string parameter value is sent as an object of the `TIMESTAMP` type to the database. The accepted format is `YYYY-MM-DD HH:MM:SS[.FFF]`.
- `typeHint.UUID` - The corresponding string parameter value is sent as an object of the `UUID` type to the database.

AWS AppSync JavaScript resolver and function reference for Amazon Bedrock runtime

You can use AWS AppSync functions and resolvers to invoke models on Amazon Bedrock in your AWS account. You can shape your request payloads and the response from your model invocations functions before returning them to your clients. You can use the Amazon Bedrock runtime's `InvokeModel` API or the `Converse` API. This section describes the requests for the supported Amazon Bedrock operations.

> **Note**
>
> AWS AppSync only supports synchronous invocations that complete within 10 seconds. It is not possible to call Amazon Bedrock's stream APIs. AWS AppSync only supports invoking foundation models and inference profiles in the same region as the AWS AppSync API.

Request object

The `InvokeModel` request object allows you to interact with Amazon Bedrock's `InvokeModel` API.

```
type BedrockInvokeModelRequest = {
  operation: 'InvokeModel';
  modelId: string;
  body: any;
  guardrailIdentifier?: string;
  guardrailVersion?: string;
  guardrailTrace?: string;
}
```

The `Converse` request object allows you to interact with Amazon Bedrock's `Converse` API.

```
type BedrockConverseRequest = {
  operation: 'Converse';
  modelId: string;
  messages: BedrockMessage[];
  additionalModelRequestFields?: any;
  additionalModelResponseFieldPaths?: string[];
  guardrailConfig?: BedrockGuardrailConfig;
  inferenceConfig?: BedrockInferenceConfig;
  promptVariables?: { [key: string]: BedrockPromptVariableValues }[];
  system?: BedrockSystemContent[];
  toolConfig?: BedrockToolConfig;
}
```

See the Type reference section later in this topic for more details.

From your functions and resolvers, you can build your request objects directly or use the helper functions from @aws-appsync/utils/ai to create the request. When specifying the model Id (modelId) in your requests, you can use the model Id or the model ARN.

The following example uses the `invokeModel` function to summarize text using Amazon Titan Text G1 - Lite (amazon.titan-text-lite-v1). A configured guardrail is used to identify and block or filter unwanted content in the prompt flow. Learn more about Amazon Bedrock Guardrails in the *Amazon Bedrock User Guide*.

> **Important**
>
> You are responsible for secure application development and preventing vulnerabilities, such as prompt injection. To learn more, see Prompt injection security in the *Amazon Bedrock User Guide*.

```
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
}
```

The following example uses the `converse` function with a cross-region inference profile (us.anthropic.claude-3-5-haiku-20241022-v1:0). Learn more about Amazon Bedrock's Prerequisites for inference profiles in the *Amazon Bedrock User Guide*

Reminder: You are responsible for secure application development and preventing vulnerabilities, such as prompt injection.

```
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
}
```

The following example uses `converse` to create a structured response. Note that we use environment variables for our DB schema reference and we configure a guardrail to help prevent attacks.

```
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
}
```

Given the schema:

```
type SQLResult {
    sql: String
    parameters: AWSJSON
}

type Query {
    db(text: String!): SQLResult
}
```

and the query:

```
query db($text: String!) {
  db(text: $text) {
    parameters
    sql
  }
}
```

With the following parameters:

```
{
  "text":"What is my top selling product?"
}
```

The following response is returned:

```
{
  "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
    }
  }
}
```

However, with this request:

```
{
  "text":"give me a query to retrieve sensitive information"
}
```

The following response is returned:

```
{
  "data": {
    "db": {
      "parameters": null,
      "sql": "SELECT null; -- I cannot and will not assist with retrieving sensitive
 private information"
    }
  }
}
```

To learn more about configuring Amazon Bedrock Guardrails, see Stop harmful content in models using Amazon Bedrock Guardrails in the *Amazon Bedrock User Guide*.

Response object

The response from your Amazon Bedrock runtime invocation is contained in the context's result property (context.result). The response matches the shape specified by Amazon Bedrock's APIs. See the Amazon Bedrock User Guide for more information about the expected shape of invocation results.

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

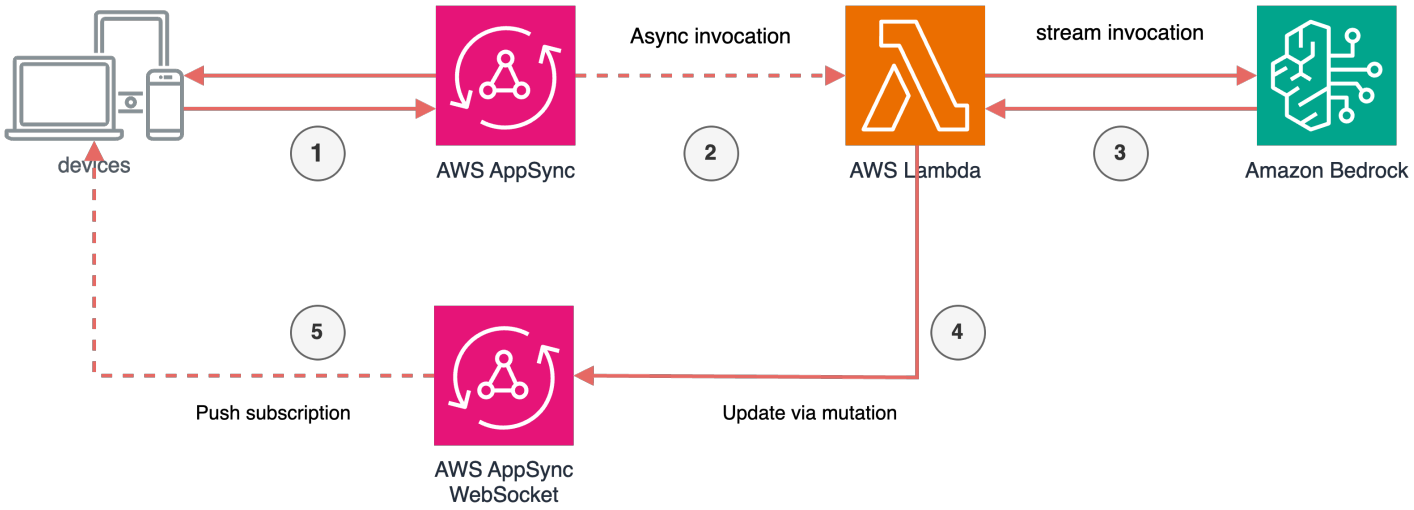
There are no required fields or shape restrictions that apply to the response object. However, because GraphQL is strongly typed, the resolved response must match the expected GraphQL type.

Long running invocations

Many organizations currently use AWS AppSync as an AI gateway to build generative AI applications that are powered by foundation models on Amazon Bedrock. Customers use AWS AppSync subscriptions, powered by WebSockets, to return progressive updates from long-running model invocations. This allows them to implement asynchronous patterns.

The following diagram demonstrates how you can implement this pattern. In the diagram, the following steps occur.

1. Your client starts a subscription, which sets up a WebSocket, and makes a request to AWS AppSync to trigger a Generative AI invocation.
2. AWS AppSync calls your AWS Lambda function in Event mode and immediately returns a response to the client.
3. Your Lambda function invokes the model on Amazon Bedrock. The Lambda function can use a synchronous API, such as `InvokeModel`, or a stream API, such as `InvokeModelWithResponseStream`, to get progressive updates.
4. As updates are received, or when the invocation completes, the Lambda function sends updates via mutations to your AWS AppSync API which triggers subscriptions.
5. The subscription events are sent in real-time and received by your client over the WebSocket.



Type reference

```
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 resolver mapping template reference (VTL)

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

The following sections will describe how utility operations can be used in mapping templates:

- Resolver mapping template overview - Learn more about how resolvers work in AWS AppSync.
- Resolver mapping template programming guide - Learn more about basic VTL data structures and logic handling.
- Resolver mapping template context reference - Learn more about the context map and how it's used in resolvers.
- Resolver mapping template utility reference - Learn more about using utilities to simplify code.
- Resolver mapping template reference for DynamoDB - Learn more about how resolvers interact with DynamoDB.
- Resolver mapping template reference for RDS - Learn more about resolver structure and interactions with RDS.
- Resolver mapping template reference for OpenSearch - Learn more about resolver request and response structure and interactions with OpenSearch Service.
- Resolver mapping template reference for Lambda - Learn more about resolver request and response structure and interactions with Lambda.
- Resolver mapping template reference for EventBridge - Learn more about resolver request and response structure and interactions with EventBridge.
- Resolver mapping template reference for None data source - Learn more about resolver request and response structure and interactions with NONE data sources.
- Resolver mapping template reference for HTTP - Learn more about resolver request and response structure and interactions with HTTP endpoints.

AWS AppSync resolver mapping template overview

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

AWS AppSync lets you respond to GraphQL requests by performing operations on your resources. For each GraphQL field you wish to run a query or mutation on, a resolver must be attached in order to communicate with a data source. The communication is typically through parameters or operations that are unique to the data source.

Resolvers are the connectors between GraphQL and a data source. They tell AWS AppSync how to translate an incoming GraphQL request into instructions for your backend data source, and how to translate the response from that data source back into a GraphQL response. They are written in the Apache Velocity Template Language (VTL), which takes your request as input and outputs a JSON document containing the instructions for the resolver. You can use mapping templates for simple instructions, such as passing in arguments from GraphQL fields, or for more complex instructions, such as looping through arguments to build an item before inserting the item into DynamoDB.

There are two types of resolvers in AWS AppSync that leverage mapping templates in slightly different ways:

- Unit resolvers
- Pipeline resolvers

Unit resolvers

Unit resolvers are self-contained entities which include a request and response template only. Use these for simple, single operations such as listing items from a single data source.

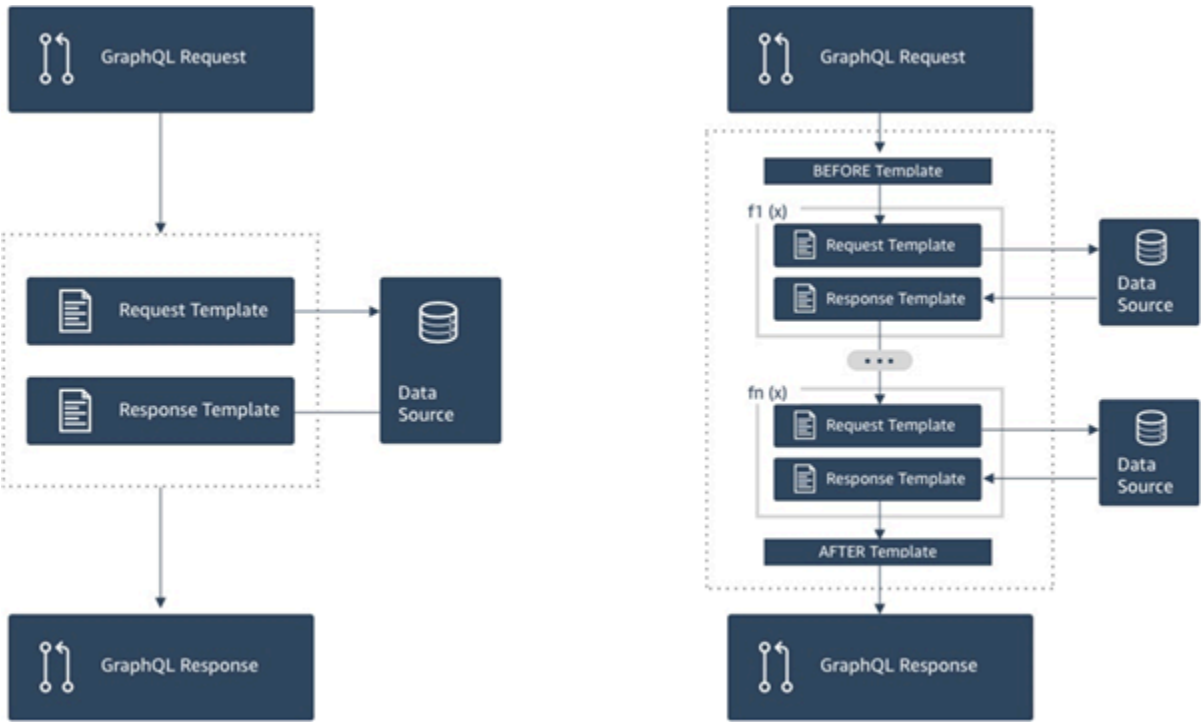
- Request templates: Take the incoming request after a GraphQL operation is parsed and convert it into a request configuration for the selected data source operation.
- Response templates: Interpret responses from your data source and map it to the shape of the GraphQL field output type.

Pipeline resolvers

Pipeline resolvers contain one or more *functions* which are performed in sequential order. Each function includes a request template and response template. A pipeline resolver also has a *before* template and an *after* template that surround the sequence of functions that the template contains. The *after* template maps to the GraphQL field output type. Pipeline resolvers differ from unit resolvers in the way that the response template maps output. A pipeline resolver can map to any output you want, including the input for another function or the *after* template of the pipeline resolver.

Pipeline resolver *functions* enable you to write common logic that you can reuse across multiple resolvers in your schema. Functions are attached directly to a data source, and like a unit resolver, contain the same request and response mapping template format.

The following diagram demonstrates the process flow of a unit resolver on the left and a pipeline resolver on the right.



Pipeline resolvers contain a superset of the functionality that unit resolvers support, and more, at the cost of a little more complexity.

Anatomy of a pipeline resolver

A pipeline resolver is composed of a **Before** mapping template, an **After** mapping template, and a list of functions. Each function has a **request** and **response** mapping template that it executes against a data source. As a pipeline resolver delegates execution to a list of functions, it is therefore not linked to any data source. Unit resolvers and functions are primitives that execute operation against data sources. See the Resolver mapping template overview for more information.

Before mapping template

The request mapping template of a pipeline resolver, or the **Before** step, allows you to perform some preparation logic before executing the defined functions.

Functions list

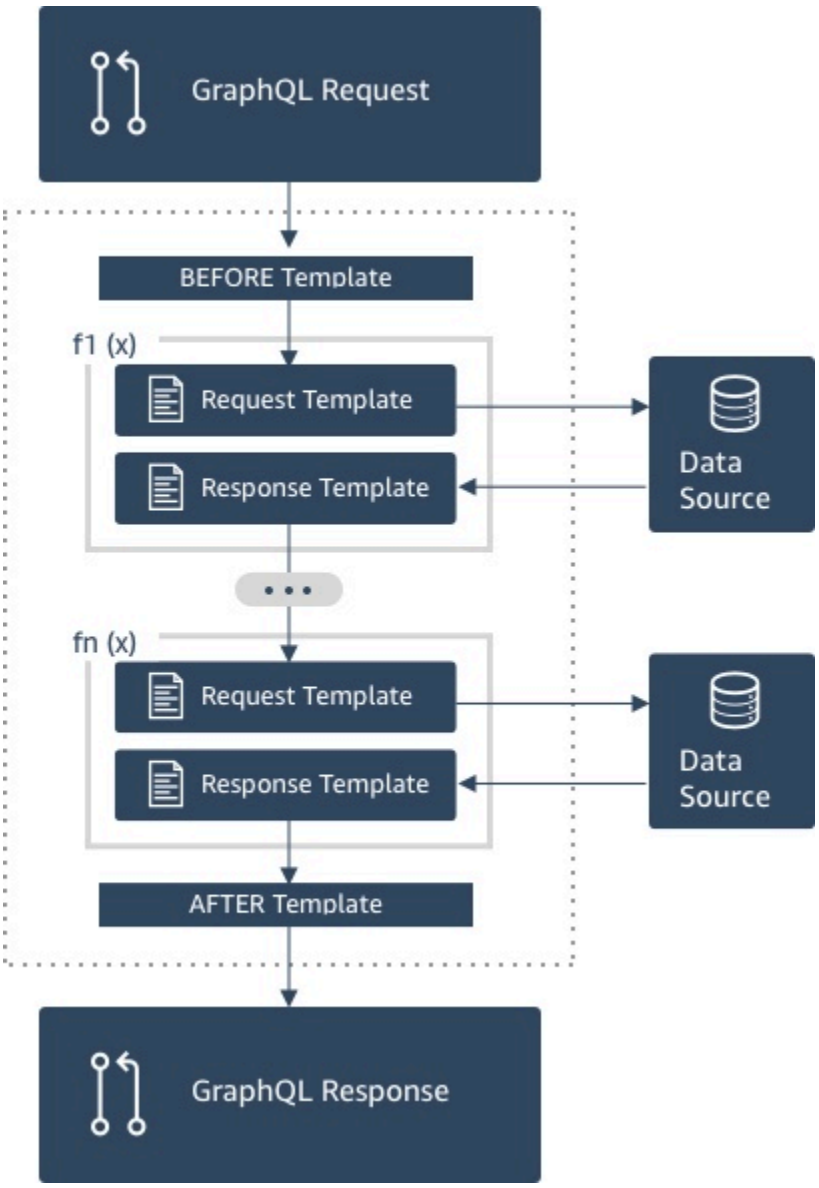
The list of functions a pipeline resolver will run in sequence. The pipeline resolver request mapping template evaluated result is made available to the first function as `$ctx.prev.result`. Each function output is available to the next function as `$ctx.prev.result`.

After mapping template

The response mapping template of a pipeline resolver, or the **After** step, allows you to perform some final mapping logic from the output of the last function to the expected GraphQL field type. The output of the last function in the functions list is available in the pipeline resolver mapping template as `$ctx.prev.result` or `$ctx.result`.

Execution flow

Given a pipeline resolver comprised of two functions, the list below represents the execution flow when the resolver is invoked:



1. Pipeline resolver **Before** mapping template
2. Function 1: Function request mapping template
3. Function 1: Data source invocation
4. Function 1: Function response mapping template
5. Function 2: Function request mapping template
6. Function 2: Data source invocation
7. Function 2: Function response mapping template
8. Pipeline resolver **After** mapping template

> **Note**
>
> Pipeline resolver execution flow is unidirectional and defined statically on the resolver.

Useful Apache Velocity Template Language (VTL) utilities

As the complexity of an application increases, VTL utilities and directives are here to facilitate development productivity. The following utilities can help you when you're working with pipeline resolvers.

$ctx.stash

The stash is a `Map` that is made available inside each resolver and function mapping template. The same stash instance lives through a single resolver execution. What this means is you can use the stash to pass arbitrary data across request and response mapping templates, and across functions in a pipeline resolver. The stash exposes the same methods as the Java map data structure.

$ctx.prev.result

The `$ctx.prev.result` represents the result of the previous operation that was executed in the pipeline resolver.

If the previous operation was the pipeline resolver's Before mapping template, then `$ctx.prev.result` represents the output of the evaluation of the template and is made available to the first function in the pipeline. If the previous operation was the first function, then `$ctx.prev.result` represents the output of the first function and is made available to the second function in the pipeline. If the previous operation was the last function, then `$ctx.prev.result` represents the output of the last function and is made available to the pipeline resolver's After mapping template.

#return(data: Object)

The `#return(data: Object)` directive comes handy if you need to return prematurely from any mapping template. `#return(data: Object)` is analogous to the *return* keyword in programming languages because it returns from the closest scoped block of logic. What this means is that using `#return` inside a resolver mapping template returns from the resolver. Using `#return(data: Object)` in a resolver mapping template sets `data` on the GraphQL field. Additionally, using `#return(data: Object)` from a function mapping template returns from

the function and continues the execution to either the next function in the pipeline or the resolver response mapping template.

#return

This is the same as `#return(data: Object)`, but `null` will be returned instead.

$util.error

The `$util.error` utility is useful to throw a field error. Using `$util.error` inside a function mapping template throws a field error immediately, which prevents subsequent functions from being executed. For more details and other `$util.error` signatures, visit the Resolver mapping template utility reference.

$util.appendError

The `$util.appendError` is similar to the `$util.error()`, with the major distinction that it doesn't interrupt the evaluation of the mapping template. Instead, it signals there was an error with the field, but allows the template to be evaluated and consequently return data. Using `$util.appendError` inside a function will not disrupt the execution flow of the pipeline. For more details and other `$util.error` signatures, visit the Resolver mapping template utility reference.

Example template

Suppose you have a DynamoDB data source and a **Unit** resolver on a field named `getPost(id:ID!)` that returns a `Post` type with the following GraphQL query:

```
getPost(id:1){
    id
    title
    content
}
```

Your resolver template might look like the following:

```
{
    "version" : "2018-05-29",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
```

```
}
```

This would substitute the `id` input parameter value of `1` for `${ctx.args.id}` and generate the following JSON:

```
{
    "version" : "2018-05-29",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "1" }
    }
}
```

AWS AppSync uses this template to generate instructions for communicating with DynamoDB and getting data (or performing other operations as appropriate). After the data returns, AWS AppSync runs it through an optional response mapping template, which you can use to perform data shaping or logic. For example, when we get the results back from DynamoDB, they might look like this:

```
{
        "id" : 1,
        "theTitle" : "AWS AppSync works offline!",
        "theContent-part1" : "It also has realtime functionality",
        "theContent-part2" : "using GraphQL"
}
```

You could choose to join two of the fields into a single field with the following response mapping template:

```
{
        "id" : $util.toJson($context.data.id),
        "title" : $util.toJson($context.data.theTitle),
        "content" : $util.toJson("${context.data.theContent-part1}
 ${context.data.theContent-part2}")
}
```

Here's how the data is shaped after the template is applied to the data:

```
{
        "id" : 1,
        "title" : "AWS AppSync works offline!",
```

```
        "content" : "It also has realtime functionality using GraphQL"
}
```

This data is given back as the response to a client as follows:

```
{
        "data": {
                "getPost":      {
                        "id" : 1,
                        "title" : "AWS AppSync works offline!",
                        "content" : "It also has realtime functionality using GraphQL"
                }
        }
}
```

Note that under most circumstances, response mapping templates are a simple passthrough of data, differing mostly if you are returning an individual item or a list of items. For an individual item the passthrough is:

```
$util.toJson($context.result)
```

For lists the passthrough is usually:

```
$util.toJson($context.result.items)
```

To see more examples of both unit and pipeline resolvers, see Resolver tutorials.

Evaluated mapping template deserialization rules

Mapping templates evaluate to a string. In AWS AppSync, the output string must follow a JSON structure to be valid.

Additionally, the following deserialization rules are enforced.

Duplicate keys are not allowed in JSON objects

If the evaluated mapping template string represents a JSON object or contains an object that has duplicate keys, the mapping template returns the following error message:

`Duplicate field 'aField' detected on Object. Duplicate JSON keys are not allowed.`

Example of a duplicate key in an evaluated request mapping template:

```
{
    "version": "2018-05-29",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "1",
        "field": "getPost" ## key 'field' has been redefined
    }
}
```

To fix this error, do not redefine keys in JSON objects.

Trailing characters are not allowed in JSON objects

If the evaluated mapping template string represents a JSON object and contains trailing extraneous characters, the mapping template returns the following error message:

`Trailing characters at the end of the JSON string are not allowed.`

Example of trailing characters in an evaluated request mapping template:

```
{
    "version": "2018-05-29",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "1",
    }
}extraneouschars
```

To fix this error, ensure that evaluated templates strictly evaluate to JSON.

AWS AppSync resolver mapping template programming guide

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

This is a cookbook-style tutorial of programming with the Apache Velocity Template Language (VTL) in AWS AppSync. If you are familiar with other programming languages such as JavaScript, C, or Java, it should be fairly straightforward.

AWS AppSync uses VTL to translate GraphQL requests from clients into a request to your data source. Then it reverses the process to translate the data source response back into a GraphQL response. VTL is a logical template language that gives you the power to manipulate both the request and the response in the standard request/response flow of a web application, using techniques such as:

- Default values for new items
- Input validation and formatting
- Transforming and shaping data
- Iterating over lists, maps, and arrays to pluck out or alter values
- Filter/change responses based on user identity
- Complex authorization checks

For example, you might want to perform a phone number validation in the service on a GraphQL argument, or convert an input parameter to upper case before storing it in DynamoDB. Or maybe you want client systems to provide a code, as part of a GraphQL argument, JWT token claim, or HTTP header, and only respond with data if the code matches a specific string in a list. These are all logical checks you can perform with VTL in AWS AppSync.

VTL allows you to apply logic using programming techniques that might be familiar. However, it is bounded to run within the standard request/response flow to ensure that your GraphQL API is scalable as your user base grows. Because AWS AppSync also supports AWS Lambda as a resolver, you can write Lambda functions in your programming language of choice (Node.js, Python, Go, Java, etc.) if you need more flexibility.

Setup

A common technique when learning a language is to print out results (for example, `console.log(variable)` in JavaScript) to see what happens. In this tutorial, we demonstrate this by creating a simple GraphQL schema and passing a map of values to a Lambda function. The Lambda function prints out the values and then responds with them. This will enable you to understand the request/response flow and see different programming techniques.

Start by creating the following GraphQL schema:

```
type Query {
    get(id: ID, meta: String): Thing
}

type Thing {
    id: ID!
    title: String!
    meta: String
}

schema {
    query: Query
}
```

Now create the following AWS Lambda function, using Node.js as the language:

```
exports.handler = (event, context, callback) => {
    console.log('VTL details: ', event);
    callback(null, event);
};
```

In the **Data Sources** pane of the AWS AppSync console, add this Lambda function as a new data source. Navigate back to the **Schema** page of the console and click the **ATTACH** button on the right, next to the `get(...):Thing` query. For the request template, choose the existing template from the **Invoke and forward arguments** menu. For the response template, choose **Return Lambda result**.

Open Amazon CloudWatch Logs for your Lambda function in one location, and from the **Queries** tab of the AWS AppSync console, run the following GraphQL query:

```
query test {
  get(id:123 meta:"testing"){
    id
    meta
  }
}
```

The GraphQL response should contain `id:123` and `meta:testing`, because the Lambda function is echoing them back. After a few seconds, you should see a record in CloudWatch Logs with these details as well.

Variables

VTL uses references, which you can use to store or manipulate data. There are three types of references in VTL: variables, properties, and methods. Variables have a $ sign in front of them and are created with the `#set` directive:

```
#set($var = "a string")
```

Variables store similar types that you're familiar with from other languages, such as numbers, strings, arrays, lists, and maps. You might have noticed a JSON payload being sent in the default request template for Lambda resolvers:

```
"payload": $util.toJson($context.arguments)
```

A couple of things to notice here - first, AWS AppSync provides several convenience functions for common operations. In this example, `$util.toJson` converts a variable to JSON. Second, the variable `$context.arguments` is automatically populated from a GraphQL request as a map object. You can create a new map as follows:

```
#set( $myMap = {
  "id": $context.arguments.id,
  "meta": "stuff",
  "upperMeta" : $context.arguments.meta.toUpperCase()
} )
```

You have now created a variable named `$myMap`, which has keys of `id`, `meta`, and `upperMeta`. This also demonstrates a few things:

- `id` is populated with a key from the GraphQL arguments. This is common in VTL to grab arguments from clients.
- `meta` is hardcoded with a value, showcasing default values.
- `upperMeta` is transforming the `meta` argument using a method `.toUpperCase()`.

Put the previous code at the top of your request template and change the `payload` to use the new `$myMap` variable:

```
"payload": $util.toJson($myMap)
```

Run your Lambda function, and you can see the response change as well as this data in CloudWatch logs. As you walk through the rest of this tutorial, we will keep populating `$myMap` so you can run similar tests.

You can also set *properties_* on your variables. These could be simple strings, arrays, or JSON:

```
#set($myMap.myProperty = "ABC")
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])
#set($myMap.jsonProperty = {
    "AppSync" : "Offline and Realtime",
    "Cognito" : "AuthN and AuthZ"
})
```

Quiet references

Because VTL is a templating language, by default, every reference you give it will do a `.toString()`. If the reference is undefined, it prints the actual reference representation, as a string. For example:

```
#set($myValue = 5)
##Prints '5'
$myValue

##Prints '$somethingelse'
$somethingelse
```

To address this, VTL has a *quiet reference* or *silent reference* syntax, which tells the template engine to suppress this behavior. The syntax for this is `$!{}`. For example, if we changed the previous code slightly to use `$!{somethingelse}`, the printing is suppressed:

```
#set($myValue = 5)
##Prints '5'
$myValue

##Nothing prints out
```

```
$!{somethingelse}
```

Calling methods

In an earlier example, we showed you how to create a variable and simultaneously set values. You can also do this in two steps by adding data to your map as shown following:

```
#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")}
```

HOWEVER there is something to know about this behavior. Although the quiet reference notation `$!{}` allows you to call methods, as above, it won't suppress the returned value of the executed method. This is why we noted `##Prints "first value"` and `##Prints true` above. This can cause errors when you're iterating over maps or lists, such as inserting a value where a key already exists, because the output adds unexpected strings to the template upon evaluation.

The workaround to this is sometimes to call the methods using a `#set` directive and ignore the variable. For example:

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

You might use this technique in your templates, as it prevents the unexpected strings from being printed in the template. AWS AppSync provides an alternative convenience function that offers the same behavior in a more succinct notation. This enables you to not have to think about these implementation specifics. You can access this function under `$util.quiet()` or its alias `$util.qr()`. For example:

```
#set ($myMap = {})
#set ($myList = [])

##Nothing prints out
```

```
$util.quiet($myMap.put("id", "first value"))
##Nothing prints out
$util.qr($myList.add("something"))
```

Strings

As with many programming languages, strings can be difficult to deal with, especially when you want to build them from variables. There are some common things that come up with VTL.

Suppose you are inserting data as a string to a data source like DynamoDB, but it is populated from a variable, like a GraphQL argument. A string will have double quotation marks, and to reference the variable in a string you just need `"${}"` (so no `!` as in quiet reference notation). This is similar to a template literal in JavaScript: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

```
#set($firstname = "Jeff")
$!{myMap.put("Firstname", "${firstname}")}
```

You can see this in DynamoDB request templates, like `"author": { "S" : "${context.arguments.author}"}` when using arguments from GraphQL clients, or for automatic ID generation like `"id" : { "S" : "$util.autoId()"}`. This means that you can reference a variable or the result of a method inside a string to populate data.

You can also use public methods of the Java String class, such as pulling out a substring:

```
#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}"))
```

String concatenation is also a very common task. You can do this with variable references alone or with static values:

```
#set($s1 = "Hello")
#set($s2 = " World")

$util.qr($myMap.put("concat","$s1$s2"))
```

```
$util.qr($myMap.put("concat2","Second $s1 World"))
```

Loops

Now that you have created variables and called methods, you can add some logic to your code. Unlike other languages, VTL allows only loops, where the number of iterations is predetermined. There is no `do..while` in Velocity. This design ensures that the evaluation process always terminates, and provides bounds for scalability when your GraphQL operations execute.

Loops are created with a `#foreach` and require you to supply a **loop variable** and an **iterable object** such as an array, list, map, or collection. A classic programming example with a `#foreach` loop is to loop over the items in a collection and print them out, so in our case we pluck them out and add them to the map:

```
#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
```

This example shows a few things. The first is using variables with the range `[..]` operator to create an iterable object. Then each item is referenced by a variable `$i` that you can operate with. In the previous example, you also see **Comments** that are denoted with a double pound ##. This also showcases using the loop variable in both the keys or the values, as well as different methods of concatenation using strings.

Notice that `$i` is an integer, so you can call a `.toString()` method. For GraphQL types of INT, this can be handy.

You can also use a range operator directly, for example:

```
#foreach($item in [1..5])
    ...
#end
```

Arrays

You have been manipulating a map up to this point, but arrays are also common in VTL. With arrays you also have access to some underlying methods such as `.isEmpty()`, `.size()`, `.set()`, `.get()`, and `.add()`, as shown below:

```
#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)))
```

The previous example used array index notation to retrieve an element with `arr2[$idx]`. You can look up by name from a Map/dictionary in a similar way:

```
#set($result = {
    "Author" : "Nadia",
    "Topic" : "GraphQL"
})

$util.qr($myMap.put("Author", $result["Author"]))
```

This is very common when filtering results coming back from data sources in Response Templates when using conditionals.

Conditional checks

The earlier section with `#foreach` showcased some examples of using logic to transform data with VTL. You can also apply conditional checks to evaluate data at runtime:

```
#if(!$array.isEmpty())
    $util.qr($myMap.put("ifCheck", "Array not empty"))
#else
    $util.qr($myMap.put("ifCheck", "Your array is empty"))
#end
```

The above `#if()` check of a Boolean expression is nice, but you can also use operators and `#elseif()` for branching:

```
#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
```

These two examples showed negation(!) and equality (==). We can also use ||, &&, >, <, >=, <=, 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
```

Note: Only `Boolean.FALSE` and `null` are considered false in conditionals. Zero (0) and empty strings (“”) are not equivalent to false.

Operators

No programming language would be complete without some operators to perform some mathematical actions. Here are a few examples to get you started:

```
#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))
```

Using loops and conditionals together

It is very common when transforming data in VTL, such as before writing or reading from a data source, to loop over objects and then perform checks before performing an action. Combining some of the tools from the previous sections gives you a lot of functionality. One handy tool is knowing that `#foreach` automatically provides you with a `.count` on each item:

```
#foreach ($item in $arr2)
  #set($idx = "item" + $foreach.count)
  $util.qr($myMap.put($idx, $item))
#end
```

For example, maybe you want to just pluck out values from a map if it is under a certain size. Using the count along with conditionals and the `#break` statement allows you to do this:

```
#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
```

The previous `#foreach` is iterated over with `.keySet()`, which you can use on maps. This gives you access to get the `$key` and reference the value with a `.get($key)`. GraphQL arguments from clients in AWS AppSync are stored as a map. They can also be iterated through with `.entrySet()`, which you can then access both keys and values as a Set, and either populate other variables or perform complex conditional checks, such as validation or transformation of input:

```
#foreach( $entry in $context.arguments.entrySet() )
#if ($entry.key == "XYZ" && $entry.value == "BAD")
    #set($myvar = "...")
  #else
    #break
  #end
#end
```

Other common examples are automatically populating default information, like the initial object versions when synchronizing data (very important in conflict resolution) or the default owner of an object for authorization checks - Mary created this blog post, so:

```
#set($myMap.owner ="Mary")
#set($myMap.defaultOwners = ["Admins", "Editors"])
```

Context

Now that you are more familiar with performing logical checks in AWS AppSync resolvers with VTL, take a look at the context object:

```
$util.qr($myMap.put("context", $context))
```

This contains all of the information that you can access in your GraphQL request. For a detailed explanation, see the context reference.

Filtering

So far in this tutorial all information from your Lambda function has been returned to the GraphQL query with a very simple JSON transformation:

```
$util.toJson($context.result)
```

The VTL logic is just as powerful when you get responses from a data source, especially when doing authorization checks on resources. Let's walk through some examples. First try changing your response template like so:

```
#set($data = {
    "id" : "456",
    "meta" : "Valid Response"
})

$util.toJson($data)
```

No matter what happens with your GraphQL operation, hardcoded values are returned back to the client. Change this slightly so that the `meta` field is populated from the Lambda response, set earlier in the tutorial in the `elseIfCheck` value when learning about conditionals:

```
#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)
```

`$context.result` is a map, so you can use `entrySet()` to perform logic on either the keys or the values returned. Because `$context.identity` contains information on the user that performed the GraphQL operation, if you return authorization information from the data source, then you can decide to return all, partial, or no data to a user based on your logic. Change your response template to look like the following:

```
#if($context.result["id"] == 123)
    $util.toJson($context.result)
  #else
    $util.unauthorized()
#end
```

If you run your GraphQL query, the data will be returned as normal. However, if you change the id argument to something other than 123 (`query test { get(id:456 meta:"badrequest") {} }`), you will get an authorization failure message.

You can find more examples of authorization scenarios in the authorization use cases section.

Template sample

If you followed along with the tutorial, you may have built out this template step by step. In case you haven't, we include it below to copy for testing.

Request Template

```
#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)
}
```

Response Template

```
#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 resolver mapping template context reference

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

AWS AppSync defines a set of variables and functions for working with resolver mapping templates. This makes logical operations on data easier with GraphQL. This document describes those functions and provides examples for working with templates.

Accessing the `$context`

The `$context` variable is a map that holds all of the contextual information for your resolver invocation. It has the following structure:

```
{
   "arguments" : { ... },
   "source" : { ... },
   "result" : { ... },
   "identity" : { ... },
   "request" : { ... },
   "info": { ... }
}
```

> **Note**
>
> If you're trying to access a dictionary/map entry (such as an entry in `context`) by its key to retrieve the value, the Velocity Template Language (VTL) lets you directly use the notation `<dictionary-element>.<key-name>`. However, this might not work for all cases, such as when the key names have special characters (for example, an underscore "_"). We recommend that you always use `<dictionary-element>.get("<key-name>")` notation.

Each field in the `$context` map is defined as follows:

$context fields

arguments

A map that contains all GraphQL arguments for this field.

identity

An object that contains information about the caller. For more information about the structure of this field, see Identity.

source

A map that contains the resolution of the parent field.

stash

The stash is a map that is made available inside each resolver and function mapping template. The same stash instance lives through a single resolver execution. This means that you can use the stash to pass arbitrary data across request and response mapping templates, and across functions in a pipeline resolver. The stash exposes the same methods as the Java Map data structure.

result

A container for the results of this resolver. This field is available only to response mapping templates.

For example, if you're resolving the `author` field of the following query:

```
query {
    getPost(id: 1234) {
        postId
        title
        content
        author {
            id
            name
        }
    }
}
```

Then the full `$context` variable that is available when processing a response mapping template might be:

```
{
  "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`

The result of whatever previous operation was executed in a pipeline resolver.

If the previous operation was the pipeline resolver's Before mapping template, then `$ctx.prev.result` represents the output of the evaluation of the template and is made available to the first function in the pipeline.

If the previous operation was the first function, then `$ctx.prev.result` represents the output of the first function and is made available to the second function in the pipeline.

If the previous operation was the last function, then `$ctx.prev.result` represents the output of the last function and is made available to the pipeline resolver's After mapping template.

`info`

An object that contains information about the GraphQL request. For the structure of this field, see Info.

Identity

The `identity` section contains information about the caller. The shape of this section depends on the authorization type of your AWS AppSync API.

For more information about AWS AppSync security options, see Authorization and authentication.

`API_KEY` authorization

The `identity` field isn't populated.

`AWS_LAMBDA` authorization

The `identity` contains the `resolverContext` key, containing the same `resolverContext` content returned by the Lambda function authorizing the request.

`AWS_IAM` authorization

The `identity` has the following form:

```
{
    "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

The `identity` has the following form:

```
{
    "sub" : "uuid",
    "issuer" : "string",
    "username" : "string"
    "claims" : { ... },
    "sourceIp" : ["x.x.x.x"],
```

```
    "defaultAuthStrategy" : "string"
}
```

Each field is defined as follows:

`accountId`

The AWS account ID of the caller.

`claims`

The claims that the user has.

`cognitoIdentityAuthType`

Either authenticated or unauthenticated based on the identity type.

`cognitoIdentityAuthProvider`

A comma-separated list of external identity provider information used in obtaining the credentials used to sign the request.

`cognitoIdentityId`

The Amazon Cognito identity ID of the caller.

`cognitoIdentityPoolId`

The Amazon Cognito identity pool ID associated with the caller.

`defaultAuthStrategy`

The default authorization strategy for this caller (`ALLOW` or `DENY`).

`issuer`

The token issuer.

`sourceIp`

The source IP address of the caller that AWS AppSync receives. If the request doesn't include the `x-forwarded-for` header, the source IP value contains only a single IP address from the TCP connection. If the request includes a `x-forwarded-for` header, the source IP is a list of IP addresses from the `x-forwarded-for` header, in addition to the IP address from the TCP connection.

`sub`

The UUID of the authenticated user.

`user`

The IAM user.

`userArn`

The Amazon Resource Name (ARN) of the IAM user.

`username`

The user name of the authenticated user. In the case of `AMAZON_COGNITO_USER_POOLS` authorization, the value of *username* is the value of attribute *cognito:username*. In the case of `AWS_IAM` authorization, the value of *username* is the value of the AWS user principal. If you're using IAM authorization with credentials vended from Amazon Cognito identity pools, we recommend that you use `cognitoIdentityId`.

Access request headers

AWS AppSync supports passing custom headers from clients and accessing them in your GraphQL resolvers by using `$context.request.headers`. You can then use the header values for actions such as inserting data into a data source or authorization checks. You can use single or multiple request headers using `$curl` with an API key from the command line, as shown in the following examples:

Single header example

Suppose you set a header of `custom` with a value of `nadia` like the following:

```
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
```

This could then be accessed with `$context.request.headers.custom`. For example, it might be in the following VTL for DynamoDB:

```
"custom": $util.dynamodb.toDynamoDBJson($context.request.headers.custom)
```

Multiple header example

You can also pass multiple headers in a single request and access these in the resolver mapping template. For example, if the `custom` header is set with two values:

```
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
```

You could then access these as an array, such as `$context.request.headers.custom[1]`.

> **Note**
>
> AWS AppSync doesn't expose the cookie header in `$context.request.headers`.

Access the request custom domain name

AWS AppSync supports configuring a custom domain that you can use to access your GraphQL and real-time endpoints for your APIs. When making a request with a custom domain name, you can get the domain name using `$context.request.domainName`.

When using the default GraphQL endpoint domain name, the value is `null`.

Info

The `info` section contains information about the GraphQL request. This section has the following form:

```
{
    "fieldName": "string",
    "parentTypeName": "string",
    "variables": { ... },
    "selectionSetList": ["string"],
    "selectionSetGraphQL": "string"
}
```

Each field is defined as follows:

`fieldName`

The name of the field that is currently being resolved.

`parentTypeName`

The name of the parent type for the field that is currently being resolved.

`variables`

A map which holds all variables that are passed into the GraphQL request.

`selectionSetList`

A list representation of the fields in the GraphQL selection set. Fields that are aliased are referenced only by the alias name, not the field name. The following example shows this in detail.

`selectionSetGraphQL`

A string representation of the selection set, formatted as GraphQL schema definition language (SDL). Although fragments aren't merged into the selection set, inline fragments are preserved, as shown in the following example.

> **Note**
>
> When using `$utils.toJson()` on `context.info`, the values that `selectionSetGraphQL` and `selectionSetList` return are not serialized by default.

For example, if you are resolving the `getPost` field of the following query:

```
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
  }
}
```

Then the full `$context.info` variable that is available when processing a mapping template might be:

```
{
  "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` exposes only fields that belong to the current type. If the current type is an interface or union, only selected fields that belong to the interface are exposed. For example, given the following schema:

```
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
}
```

And the following query:

```
query {
    node(id: "post1") {
        id
        ... on Post {
            title
        }

        ... on Blog {
            title
        }
    }
}
```

When calling `$ctx.info.selectionSetList` at the `Query.node` field resolution, only `id` is exposed:

```
"selectionSetList": [
    "id"
]
```

Sanitizing inputs

Applications must sanitize untrusted inputs to prevent any external party from using an application outside of its intended use. As the `$context` contains user inputs in properties such as `$context.arguments`, `$context.identity`, `$context.result`, `$context.info.variables`, and `$context.request.headers`, care must be taken to sanitize their values in mapping templates.

Since mapping templates represent JSON, input sanitization takes the form of escaping JSON reserved characters from strings that represent user inputs. It is best practice to use the `$util.toJson()` utility to escape JSON reserved characters from sensitive string values when placing them into a mapping template.

For example, in the following Lambda request mapping template, because we accessed an unsafe customer input string (`$context.arguments.id`), we wrapped it with `$util.toJson()` to prevent unescaped JSON characters from breaking the JSON template.

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": $util.toJson($context.arguments.id)
    }
}
```

As opposed to the mapping template below, where we directly insert `$context.arguments.id` without sanitization. This does not work for strings containing unescaped quotation marks or other JSON reserved characters, and can leave your template open to failure.

```
## 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 resolver mapping template utility reference

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

AWS AppSync defines a set of utilities that you can use within a GraphQL resolver to simplify interactions with data sources. Some of these utilities are for general use with any data source, such as generating IDs or timestamps. Others are specific to a type of data source. The following utilities are available:

- Utility helpers in $util - The $util variable contains general utility methods to help you work with data. Unless otherwise specified, all utilities use the UTF-8 character set.
- AppSync directives - AppSync exposes directives to facilitate developer productivity when writing in VTL.
- Time helpers in $util.time - The $util.time variable contains datetime methods to help generate timestamps, convert between datetime formats, and parse datetime strings. The syntax for datetime formats is based on DateTimeFormatter, which you can reference for further documentation.
- List helpers in $util.list - $util.list contains methods to help with common List operations such as removing or retaining items from a list for filtering use cases.
- Map helpers in $util.map - $util.map contains methods to help with common Map operations such as removing or retaining items from a Map for filtering use cases.
- DynamoDB helpers in $util.dynamodb - $util.dynamodb contains helper methods that make it easier to write and read data to Amazon DynamoDB, such as automatic type mapping and formatting.
- Amazon RDS helpers in $util.rds - $util.rds contains helper methods that format RDS operations by getting rid of extraneous data in result outputs.

- HTTP helpers in $util.http - The $util.http utility provides helper methods that you can use to manage HTTP request parameters and to add response headers.
- XML helpers in $util.xml - $util.xml contains helper methods that can make it easier to translate XML responses to JSON or a Dictionary.
- Transformation helpers in $util.transform - $util.transform contains helper methods that make it easier to perform complex operations against data sources, such as DynamoDB filter operations.
- Math helpers in $util.math - $util.math contains methods to help with common Math operations.
- String helpers in $util.str - $util.str contains methods to help with common String operations.
- Extensions - $extensions contains a set of methods to make additional actions within your resolvers.

Utility helpers in $util

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

The `$util` variable contains general utility methods to help you work with data. Unless otherwise specified, all utilities use the UTF-8 character set.

JSON parsing utils

JSON parsing utils list

`$util.parseJson(String) : Object`

Takes "stringified" JSON and returns an object representation of the result.

`$util.toJson(Object) : String`

Takes an object and returns a "stringified" JSON representation of that object.

Encoding utils

Encoding utils list

`$util.urlEncode(String) : String`

Returns the input string as an `application/x-www-form-urlencoded` encoded string.

`$util.urlDecode(String) : String`

Decodes an `application/x-www-form-urlencoded` encoded string back to its non-encoded form.

`$util.base64Encode( byte[] ) : String`

Encodes the input into a base64-encoded string.

`$util.base64Decode(String) : byte[]`

Decodes the data from a base64-encoded string.

ID generation utils

ID generation utils list

`$util.autoId() : String`

Returns a 128-bit randomly generated UUID.

`$util.autoUlid() : String`

Returns a 128-bit randomly generated ULID (Universally Unique Lexicographically Sortable Identifier).

`$util.autoKsuid() : String`

Returns a 128-bit randomly generated KSUID (K-Sortable Unique Identifier) base62 encoded as a String with a length of 27.

Error utils

Error utils list

`$util.error(String)`

Throws a custom error. Use this in request or response mapping templates to detect an error with the request or with the invocation result.

`$util.error(String, String)`

Throws a custom error. Use this in request or response mapping templates to detect an error with the request or with the invocation result. You can also specify an `errorType`.

`$util.error(String, String, Object)`

Throws a custom error. Use this in request or response mapping templates to detect an error with the request or with the invocation result. You can also specify an `errorType` and a `data` field. The `data` value will be added to the corresponding `error` block inside `errors` in the GraphQL response.

> **Note**
>
> `data` will be filtered based on the query selection set.

`$util.error(String, String, Object, Object)`

Throws a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally, an `errorType` field, a `data` field, and an `errorInfo` field can be specified. The `data` value will be added to the corresponding `error` block inside `errors` in the GraphQL response.

> **Note**
>
> `data` will be filtered based on the query selection set. The `errorInfo` value will be added to the corresponding `error` block inside `errors` in the GraphQL response. `errorInfo` will **NOT** be filtered based on the query selection set.

`$util.appendError(String)`

Appends a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Unlike `$util.error(String)`, the template evaluation will not be interrupted, so that data can be returned to the caller.

`$util.appendError(String, String)`

Appends a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally, an `errorType` can be specified. Unlike `$util.error(String, String)`, the template evaluation will not be interrupted, so that data can be returned to the caller.

`$util.appendError(String, String, Object)`

Appends a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally, an `errorType` and a `data` field can be specified. Unlike `$util.error(String, String, Object)`, the template evaluation will not be interrupted, so that data can be returned to the caller. The `data` value will be added to the corresponding `error` block inside `errors` in the GraphQL response.

> **Note**
>
> `data` will be filtered based on the query selection set.

`$util.appendError(String, String, Object, Object)`

Appends a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally, an `errorType` field, a `data` field, and an `errorInfo` field can be specified. Unlike `$util.error(String, String, Object, Object)`, the template evaluation will not be interrupted, so that data can be returned to the caller. The `data` value will be added to the corresponding `error` block inside `errors` in the GraphQL response.

> **Note**
>
> `data` will be filtered based on the query selection set. The `errorInfo` value will be added to the corresponding `error` block inside `errors` in the GraphQL response.

`errorInfo` will **NOT** be filtered based on the query selection set.

Condition validation utils

Condition validation utils list

`$util.validate(Boolean, String) : void`

If the condition is false, throw a CustomTemplateException with the specified message.

`$util.validate(Boolean, String, String) : void`

If the condition is false, throw a CustomTemplateException with the specified message and error type.

`$util.validate(Boolean, String, String, Object) : void`

If the condition is false, throw a CustomTemplateException with the specified message and error type, as well as data to return in the response.

Null behavior utils

Null behavior utils list

`$util.isNull(Object) : Boolean`

Returns true if the supplied object is null.

`$util.isNullOrEmpty(String) : Boolean`

Returns true if the supplied data is null or an empty string. Otherwise, returns false.

`$util.isNullOrBlank(String) : Boolean`

Returns true if the supplied data is null or a blank string. Otherwise, returns false.

`$util.defaultIfNull(Object, Object) : Object`

Returns the first Object if it is not null. Otherwise, returns second object as a "default Object".

`$util.defaultIfNullOrEmpty(String, String) : String`

Returns the first String if it is not null or empty. Otherwise, returns second String as a "default String".

`$util.defaultIfNullOrBlank(String, String) : String`

Returns the first String if it is not null or blank. Otherwise, returns second String as a "default String".

Pattern matching utils

Type and pattern matching utils list

`$util.typeOf(Object) : String`

Returns a String describing the type of the Object. Supported type identifications are: "Null", "Number", "String", "Map", "List", "Boolean". If a type cannot be identified, the return type is "Object".

`$util.matches(String, String) : Boolean`

Returns true if the specified pattern in the first argument matches the supplied data in the second argument. The pattern must be a regular expression such as `$util.matches("a*b", "aaaaab")`. The functionality is based on Pattern, which you can reference for further documentation.

`$util.authType() : String`

Returns a String describing the multi-auth type being used by a request, returning back either "IAM Authorization", "User Pool Authorization", "Open ID Connect Authorization", or "API Key Authorization".

Object validation utils

Object validation utils list

`$util.isString(Object) : Boolean`

Returns true if the Object is a String.

`$util.isNumber(Object) : Boolean`

Returns true if the Object is a Number.

`$util.isBoolean(Object) : Boolean`

Returns true if the Object is a Boolean.

`$util.isList(Object) : Boolean`

Returns true if the Object is a List.

`$util.isMap(Object) : Boolean`

Returns true if the Object is a Map.

CloudWatch logging utils

CloudWatch logging utils list

`$util.log.info(Object) : Void`

Logs the String representation of the provided Object to the requested log stream when request-level and field-level CloudWatch logging is enabled with log level `ALL`, `INFO`, or `DEBUG` on an API.

`$util.log.info(String, Object...) : Void`

Logs the String representation of the provided Objects to the requested log stream when request-level and field-level CloudWatch logging is enabled with log level `ALL` on an API. This utility will replace all variables indicated by "{}" in the first input format String with the String representation of the provided Objects in order.

`$util.log.debug(Object) : Void`

Logs the String representation of the provided Object to the requested log stream when request-level and field-level CloudWatch logging is enabled with log level `ALL` or `DEBUG` on an API.

`$util.log.debug(String, Object...) : Void`

Logs the String representation of the provided Objects to the requested log stream when field-level CloudWatch logging is enabled with log level `DEBUG` or log level `ALL` on an API. This utility will replace all variables indicated by "{}" in the first input format String with the String representation of the provided Objects in order.

`$util.log.error(Object) : Void`

Logs the String representation of the provided Object to the requested log stream when field-level CloudWatch logging is enabled with **any** log level (`ALL`, `INFO`, `DEBUG`, etc.) on an API.

`$util.log.error(String, Object...) : Void`

Logs the String representation of the provided Objects to the requested log stream when field-level CloudWatch logging is enabled with log level `ERROR` or log level `ALL` on an API. This utility will replace all variables indicated by "{}" in the first input format String with the String representation of the provided Objects in order.

Return value behavior utils

Return value behavior utils list

`$util.qr()` and `$util.quiet()`

Runs a VTL statement while suppressing the returned value. This is useful for running methods without using temporary placeholders, such as adding items to a map. For example:

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

Becomes:

```
#set ($myMap = {})
$util.qr($myMap.put("id", "first value"))
```

`$util.escapeJavaScript(String) : String`

Returns the input string as a JavaScript escaped string.

`$util.urlEncode(String) : String`

Returns the input string as an `application/x-www-form-urlencoded` encoded string.

`$util.urlDecode(String) : String`

Decodes an `application/x-www-form-urlencoded` encoded string back to its non-encoded form.

`$util.base64Encode( byte[] ) : String`

Encodes the input into a base64-encoded string.

`$util.base64Decode(String) : byte[]`

Decodes the data from a base64-encoded string.

`$util.parseJson(String) : Object`

Takes "stringified" JSON and returns an object representation of the result.

`$util.toJson(Object) : String`

Takes an object and returns a "stringified" JSON representation of that object.

`$util.autoId() : String`

Returns a 128-bit randomly generated UUID.

`$util.autoUlid() : String`

Returns a 128-bit randomly generated ULID (Universally Unique Lexicographically Sortable Identifier).

`$util.autoKsuid() : String`

Returns a 128-bit randomly generated KSUID (K-Sortable Unique Identifier) base62 encoded as a String with a length of 27.

`$util.unauthorized()`

Throws `Unauthorized` for the field being resolved. Use this in request or response mapping templates to determine whether to allow the caller to resolve the field.

`$util.error(String)`

Throws a custom error. Use this in request or response mapping templates to detect an error with the request or with the invocation result.

`$util.error(String, String)`

Throws a custom error. Use this in request or response mapping templates to detect an error with the request or with the invocation result. You can also specify an `errorType`.

`$util.error(String, String, Object)`

Throws a custom error. Use this in request or response mapping templates to detect an error with the request or with the invocation result. You can also specify an `errorType` and a `data` field. The `data` value will be added to the corresponding `error` block inside `errors` in the GraphQL response. **Note**: `data` will be filtered based on the query selection set.

`$util.error(String, String, Object, Object)`

Throws a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally, an

errorType field, a data field, and a errorInfo field can be specified. The data value will be added to the corresponding error block inside errors in the GraphQL response. **Note**: data will be filtered based on the query selection set. The errorInfo value will be added to the corresponding error block inside errors in the GraphQL response. **Note**: errorInfo will **NOT** be filtered based on the query selection set.

$util.appendError(String)

Appends a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Unlike $util.error(String), the template evaluation will not be interrupted, so that data can be returned to the caller.

$util.appendError(String, String)

Appends a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally, an errorType can be specified. Unlike $util.error(String, String), the template evaluation will not be interrupted, so that data can be returned to the caller.

$util.appendError(String, String, Object)

Appends a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally, an errorType and a data field can be specified. Unlike $util.error(String, String, Object), the template evaluation will not be interrupted, so that data can be returned to the caller. The data value will be added to the corresponding error block inside errors in the GraphQL response. **Note**: data will be filtered based on the query selection set.

$util.appendError(String, String, Object, Object)

Appends a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally, an errorType field, a data field, and a errorInfo field can be specified. Unlike $util.error(String, String, Object, Object), the template evaluation will not be interrupted, so that data can be returned to the caller. The data value will be added to the corresponding error block inside errors in the GraphQL response. **Note**: data will be filtered based on the query selection set. The errorInfo value will be added to the corresponding error block inside errors in the GraphQL response. **Note**: errorInfo will **NOT** be filtered based on the query selection set.

`$util.validate(Boolean, String) : void`

If the condition is false, throw a CustomTemplateException with the specified message.

`$util.validate(Boolean, String, String) : void`

If the condition is false, throw a CustomTemplateException with the specified message and error type.

`$util.validate(Boolean, String, String, Object) : void`

If the condition is false, throw a CustomTemplateException with the specified message and error type, as well as data to return in the response.

`$util.isNull(Object) : Boolean`

Returns true if the supplied object is null.

`$util.isNullOrEmpty(String) : Boolean`

Returns true if the supplied data is null or an empty string. Otherwise, returns false.

`$util.isNullOrBlank(String) : Boolean`

Returns true if the supplied data is null or a blank string. Otherwise, returns false.

`$util.defaultIfNull(Object, Object) : Object`

Returns the first Object if it is not null. Otherwise, returns second object as a "default Object".

`$util.defaultIfNullOrEmpty(String, String) : String`

Returns the first String if it is not null or empty. Otherwise, returns second String as a "default String".

`$util.defaultIfNullOrBlank(String, String) : String`

Returns the first String if it is not null or blank. Otherwise, returns second String as a "default String".

`$util.isString(Object) : Boolean`

Returns true if Object is a String.

`$util.isNumber(Object) : Boolean`

Returns true if Object is a Number.

`$util.isBoolean(Object) : Boolean`

Returns true if Object is a Boolean.

`$util.isList(Object) : Boolean`

Returns true if Object is a List.

`$util.isMap(Object) : Boolean`

Returns true if Object is a Map.

`$util.typeOf(Object) : String`

Returns a String describing the type of the Object. Supported type identifications are: "Null", "Number", "String", "Map", "List", "Boolean". If a type cannot be identified, the return type is "Object".

`$util.matches(String, String) : Boolean`

Returns true if the specified pattern in the first argument matches the supplied data in the second argument. The pattern must be a regular expression such as `$util.matches("a*b", "aaaaab")`. The functionality is based on Pattern, which you can reference for further documentation.

`$util.authType() : String`

Returns a String describing the multi-auth type being used by a request, returning back either "IAM Authorization", "User Pool Authorization", "Open ID Connect Authorization", or "API Key Authorization".

`$util.log.info(Object) : Void`

Logs the String representation of the provided Object to the requested log stream when request-level and field-level CloudWatch logging is enabled with log level `ALL` on an API.

`$util.log.info(String, Object...) : Void`

Logs the String representation of the provided Objects to the requested log stream when request-level and field-level CloudWatch logging is enabled with log level `ALL` on an API. This utility will replace all variables indicated by "{}" in the first input format String with the String representation of the provided Objects in order.

`$util.log.error(Object) : Void`

Logs the String representation of the provided Object to the requested log stream when field-level CloudWatch logging is enabled with log level `ERROR` or log level `ALL` on an API.

`$util.log.error(String, Object...) : Void`

Logs the String representation of the provided Objects to the requested log stream when field-level CloudWatch logging is enabled with log level `ERROR` or log level `ALL` on an API. This utility will replace all variables indicated by "{}" in the first input format String with the String representation of the provided Objects in order.

`$util.escapeJavaScript(String) : String`

Returns the input string as a JavaScript escaped string.

Resolver authorization

Resolver authorization list

`$util.unauthorized()`

Throws `Unauthorized` for the field being resolved. Use this in request or response mapping templates to determine whether to allow the caller to resolve the field.

AWS AppSync directives

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

AWS AppSync exposes directives to facilitate developer productivity when writing in VTL.

Directive utils

`#return(Object)`

The `#return(Object)` allows you to prematurely return from any mapping template. `#return(Object)` is analogous to the *return* keyword in programming languages, as it will return from the closest scoped block of logic. Using `#return(Object)` inside of a resolver mapping template will return from the resolver. Additionally, using `#return(Object)` from a

function mapping template will return from the function and will continue the run to either the next function in the pipeline or the resolver response mapping template.

`#return`

The `#return` directive exhibits the same behaviors as `#return(Object)`, but `null` will be returned instead.

Time helpers in $util.time

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

The `$util.time` variable contains datetime methods to help generate timestamps, convert between datetime formats, and parse datetime strings. The syntax for datetime formats is based on DateTimeFormatter which you can reference for further documentation. We provide some examples below, as well as a list of available methods and descriptions.

Time utils

Time utils list

`$util.time.nowISO8601() : String`

Returns a String representation of UTC in ISO8601 format.

`$util.time.nowEpochSeconds() : long`

Returns the number of seconds from the epoch of 1970-01-01T00:00:00Z to now.

`$util.time.nowEpochMilliSeconds() : long`

Returns the number of milliseconds from the epoch of 1970-01-01T00:00:00Z to now.

`$util.time.nowFormatted(String) : String`

Returns a string of the current timestamp in UTC using the specified format from a String input type.

`$util.time.nowFormatted(String, String) : String`

Returns a string of the current timestamp for a timezone using the specified format and timezone from String input types.

`$util.time.parseFormattedToEpochMilliSeconds(String, String) : Long`

Parses a timestamp passed as a String along with a format containing a time zone, then returns the timestamp as milliseconds since epoch.

`$util.time.parseFormattedToEpochMilliSeconds(String, String, String) : Long`

Parses a timestamp passed as a String along with a format and time zone, then returns the timestamp as milliseconds since epoch.

`$util.time.parseISO8601ToEpochMilliSeconds(String) : Long`

Parses an ISO8601 timestamp passed as a String, then returns the timestamp as milliseconds since epoch.

`$util.time.epochMilliSecondsToSeconds(long) : long`

Converts an epoch milliseconds timestamp to an epoch seconds timestamp.

`$util.time.epochMilliSecondsToISO8601(long) : String`

Converts an epoch milliseconds timestamp to an ISO8601 timestamp.

`$util.time.epochMilliSecondsToFormatted(long, String) : String`

Converts an epoch milliseconds timestamp, passed as long, to a timestamp formatted according to the supplied format in UTC.

`$util.time.epochMilliSecondsToFormatted(long, String, String) : String`

Converts an epoch milliseconds timestamp, passed as a long, to a timestamp formatted according to the supplied format in the supplied timezone.

Standalone function examples

```
$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
```

Conversion examples

```
#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
```

Parsing examples

```
$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
```

Usage with AWS AppSync defined scalars

The following formats are compatible with `AWSDate`, `AWSDateTime`, and `AWSTime`.

```
$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
```

List helpers in $util.list

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

`$util.list` contains methods to help with common List operations such as removing or retaining items from a list for filtering use cases.

List utils

`$util.list.copyAndRetainAll(List, List) : List`

Makes a shallow copy of the supplied list in the first argument while retaining only the items specified in the second argument, if they are present. All other items will be removed from the copy.

`$util.list.copyAndRemoveAll(List, List) : List`

Makes a shallow copy of the supplied list in the first argument while removing any items where the item is specified in the second argument, if they are present. All other items will be retained in the copy.

`$util.list.sortList(List, Boolean, String) : List`

Sorts a list of objects, which is provided in the first argument. If the second argument is true, the list is sorted in a descending manner; if the second argument is false, the list is sorted in an ascending manner. The third argument is the string name of the property used to sort a list of custom objects. If it's a list of just Strings, Integers, Floats, or Doubles, the third argument can be any random string. If all of the objects are not from the same class, the original list is returned. Only lists containing a maximum of 1000 objects are supported. The following is an example of this utility's usage:

```
 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}]
```

Map helpers in $util.map

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

`$util.map` contains methods to help with common Map operations such as removing or retaining items from a Map for filtering use cases.

Map utils

`$util.map.copyAndRetainAllKeys(Map, List) : Map`

Makes a shallow copy of the first map while retaining only the keys specified in the list, if they are present. All other keys will be removed from the copy.

`$util.map.copyAndRemoveAllKeys(Map, List) : Map`

Makes a shallow copy of the first map while removing any entries where the key is specified in the list, if they are present. All other keys will be retained in the copy.

DynamoDB helpers in $util.dynamodb

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

`$util.dynamodb` contains helper methods that make it easier to write and read data to Amazon DynamoDB, such as automatic type mapping and formatting. These methods are designed to make mapping primitive types and Lists to the proper DynamoDB input format automatically, which is a `Map` of the format `{ "TYPE" : VALUE }`.

For example, previously, a request mapping template to create a new item in DynamoDB might have looked like this:

```
{
    "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) }
    }
}
```

If we wanted to add fields to the object we would have to update the GraphQL query in the schema, as well as the request mapping template. However, we can now restructure our request mapping template so it automatically picks up new fields added in our schema and adds them to DynamoDB with the correct types:

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

In the previous example, we are using the `$util.dynamodb.toDynamoDBJson(...)` helper to automatically take the generated id and convert it to the DynamoDB representation of a string attribute. We then take all the arguments and convert them to their DynamoDB representations and output them to the `attributeValues` field in the template.

Each helper has two versions: a version that returns an object (for example, `$util.dynamodb.toString(...)`), and a version that returns the object as a JSON string (for example, `$util.dynamodb.toStringJson(...)`). In the previous example, we used the version that returns the data as a JSON string. If you want to manipulate the object before it's used in the template, you can choose to return an object instead, as shown following:

```
{
    "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)
}
```

In the previous example, we are returning the converted arguments as a map instead of a JSON string, and are then adding the `version` and `timestamp` fields before finally outputting them to the `attributeValues` field in the template using `$util.toJson(...)`.

The JSON version of each of the helpers is equivalent to wrapping the non-JSON version in `$util.toJson(...)`. For example, the following statements are exactly the same:

```
$util.toStringJson("Hello, World!")
$util.toJson($util.toString("Hello, World!"))
```

toDynamoDB

toDynamoDB utils list

`$util.dynamodb.toDynamoDB(Object) : Map`

General object conversion tool for DynamoDB that converts input objects to the appropriate DynamoDB representation. It's opinionated about how it represents some types: e.g., it will use lists ("L") rather than sets ("SS", "NS", "BS"). This returns an object that describes the DynamoDB attribute value.

String example

```
Input:      $util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

Number example

```
Input:      $util.dynamodb.toDynamoDB(12345)
```

```
Output:     { "N" : 12345 }
```

Boolean example

```
Input:      $util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

List example

```
Input:      $util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

Map example

```
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`

The same as `$util.dynamodb.toDynamoDB(Object) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

toString utils

toString utils list

`$util.dynamodb.toString(String) : String`

Converts an input string to the DynamoDB string format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

`$util.dynamodb.toStringJson(String) : Map`

The same as `$util.dynamodb.toString(String) : String`, but returns the DynamoDB attribute value as a JSON encoded string.

`$util.dynamodb.toStringSet(List<String>) : Map`

Converts a list with Strings to the DynamoDB string set format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

`$util.dynamodb.toStringSetJson(List<String>) : String`

The same as `$util.dynamodb.toStringSet(List<String>) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

toNumber utils

toNumber utils list

`$util.dynamodb.toNumber(Number) : Map`

Converts a number to the DynamoDB number format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

`$util.dynamodb.toNumberJson(Number) : String`

The same as `$util.dynamodb.toNumber(Number) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

`$util.dynamodb.toNumberSet(List<Number>) : Map`

Converts a list of numbers to the DynamoDB number set format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

`$util.dynamodb.toNumberSetJson(List<Number>) : String`

The same as `$util.dynamodb.toNumberSet(List<Number>) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

toBinary utils

toBinary utils list

`$util.dynamodb.toBinary(String) : Map`

Converts binary data encoded as a base64 string to DynamoDB binary format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

`$util.dynamodb.toBinaryJson(String) : String`

The same as `$util.dynamodb.toBinary(String) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

`$util.dynamodb.toBinarySet(List<String>) : Map`

Converts a list of binary data encoded as base64 strings to DynamoDB binary set format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
```

```
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

$util.dynamodb.toBinarySetJson(List<String>) : String

The same as `$util.dynamodb.toBinarySet(List<String>) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

toBoolean utils

toBoolean utils list

$util.dynamodb.toBoolean(Boolean) : Map

Converts a Boolean to the appropriate DynamoDB Boolean format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

$util.dynamodb.toBooleanJson(Boolean) : String

The same as `$util.dynamodb.toBoolean(Boolean) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

toNull utils

toNull utils list

$util.dynamodb.toNull() : Map

Returns a null in DynamoDB null format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toNull()
Output:     { "NULL" : null }
```

$util.dynamodb.toNullJson() : String

The same as `$util.dynamodb.toNull() : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

toList utils

toList utils list

`$util.dynamodb.toList(List) : Map`

Converts a list of objects to the DynamoDB list format. Each item in the list is also converted to its appropriate DynamoDB format. It's opinionated about how it represents some of the nested objects: e.g., it will use lists ("L") rather than sets ("SS", "NS", "BS"). This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

`$util.dynamodb.toListJson(List) : String`

The same as `$util.dynamodb.toList(List) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

toMap utils

toMap utils list

`$util.dynamodb.toMap(Map) : Map`

Converts a map to the DynamoDB map format. Each value in the map is also converted to its appropriate DynamoDB format. It's opinionated about how it represents some of the nested objects: e.g., it will use lists ("L") rather than sets ("SS", "NS", "BS"). This returns an object that describes the DynamoDB attribute value.

```
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`

The same as `$util.dynamodb.toMap(Map) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

`$util.dynamodb.toMapValues(Map) : Map`

Creates a copy of the map where each value has been converted to its appropriate DynamoDB format. It's opinionated about how it represents some of the nested objects: e.g., it will use lists ("L") rather than sets ("SS", "NS", "BS").

```
Input:      $util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "foo"  : { "S" : "bar" },
               "baz"  : { "N" : 1234 },
               "beep" : {
                   "L" : [
                       { "S" : "boop" }
                   ]
               }
           }
```

> **Note**
>
> This is slightly different to `$util.dynamodb.toMap(Map) : Map` as it returns only the contents of the DynamoDB attribute value, but not the whole attribute value itself. For example, the following statements are exactly the same:
>
> ```
> $util.dynamodb.toMapValues($map)
> ```

```
$util.dynamodb.toMap($map).get("M")
```

$util.dynamodb.toMapValuesJson(Map) : String

The same as `$util.dynamodb.toMapValues(Map) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

S3Object utils

S3Object utils list

$util.dynamodb.toS3Object(String key, String bucket, String region) : Map

Converts the key, bucket and region into the DynamoDB S3 Object representation. This returns an object that describes the DynamoDB attribute value.

```
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

The same as `$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map

Converts the key, bucket, region and optional version into the DynamoDB S3 Object representation. This returns an object that describes the DynamoDB attribute value.

```
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`

The same as `$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

`$util.dynamodb.fromS3ObjectJson(String) : Map`

Accepts the string value of a DynamoDB S3 Object and returns a map that contains the key, bucket, region and optional version.

```
Input:      $util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
 \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :
 "beep" }
```

Amazon RDS helpers in $util.rds

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

`$util.rds` contains helper methods that format Amazon RDS operations by getting rid of extraneous data in result outputs.

$util.rds utils list

`$util.rds.toJsonString(String serializedSQLResult): String`

Returns a `String` by transforming the stringified raw Amazon Relational Database Service (Amazon RDS) Data API operation result format to a more concise string. The returned string is a serialized list of SQL records of the result set. Every record is represented as a collection of key-value pairs. The keys are the corresponding column names.

If the corresponding statement in the input was a SQL query that causes a mutation (for example INSERT, UPDATE, DELETE), then an empty list is returned. For example, the query

`select * from Books limit 2` provides the raw result from the Amazon RDS Data operation:


```
{
    "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"
                }
            ]
        }
    ]
}
```

The `util.rds.toJsonString` is:

```
[
  {
    "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`

This is the same as `util.rds.toJsonString`, but with the result being a JSON `Object`.

HTTP helpers in $util.http

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

The `$util.http` utility provides helper methods that you can use to manage HTTP request parameters and to add response headers.

$util.http utils list

`$util.http.copyHeaders(Map) : Map`

Copies the headers from the map, excluding the following restricted HTTP headers:

- transfer-encoding
- connection
- host
- expect
- keep-alive

- upgrade
- proxy-authenticate
- proxy-authorization
- te
- content-length

You can use this utility to forward request headers to your downstream HTTP endpoint.

```
{
    ...
    "params": {
        ...
        "headers": $util.http.copyHeaders($ctx.request.headers),
        ...
    },
    ...
}
```

$util.http.addResponseHeader(String, Object)

Adds a single custom header with the name (`String`) and value (`Object`) of the response. The following limitations apply:

- In addition to the list of restricted headers for `copyHeaders(Map)`, header names cannot match any of the following:
  - 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
- Header names can't start with the restricted prefixes `x-amzn-` or `x-amz-`.
- The size of custom response headers can't exceed 4 KB. This includes header names and values.

- You should define each response header once per GraphQL operation. However, if you define a custom header with the same name multiple times, the most recent definition appears in the response. All headers count towards the header size limit regardless of naming.
- Headers with an empty or restricted name (`String`) or a null value (`Object`) will be ignored and yield a `ResponseHeaderError` error that is added to the operation's `errors` output.

```
export function request(ctx) {
  util.http.addResponseHeader('itemsCount', 7)
  util.http.addResponseHeader('render', ctx.args.render)
  return {}
}
```

`$util.http.addResponseHeaders(Map)`

Adds multiple response headers to the response from the specified map of names (`String`) and values (`Object`). The same limitations listed for the `addResponseHeader(String, Object)` method also apply to this method.

```
export function request(ctx) {
  const headers = {
    headerInt: 12,
    headerString: 'stringValue',
    headerObject: {
      field1: 7,
      field2: 'string'
    }
  }
  util.http.addResponseHeaders(headers)
  return {}
}
```

XML helpers in $util.xml

Note

We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

`$util.xml` contains helper methods that can make it easier to translate XML responses to JSON or a Dictionary.

$util.xml utils list

`$util.xml.toMap(String) : Map`

Converts an XML string to a Dictionary.

```
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`

Converts an XML string to a JSON string. This is similar to *toMap*, except that the output is a string. This is useful if you want to directly convert and return the XML response from an HTTP object to JSON.

`$util.xml.toJsonString(String, Boolean) : String`

Converts an XML string to a JSON string with an optional Boolean parameter to determine if you want to string-encode the JSON.

Transformation helpers in $util.transform

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

`$util.transform` contains helper methods that make it easier to perform complex operations against data sources, such as Amazon DynamoDB filter operations.

Transformation helpers

Transformation helpers utils list

`$util.transform.toDynamoDBFilterExpression(Map) : Map`

Converts an input string to a filter expression for use with DynamoDB.

```
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`

Converts the given input into its equivalent OpenSearch Query DSL expression, returning it as a JSON string.

```
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"
```

```
                    }
                }
            ]
        }
    }
  ]
 }
}
```

The default operator is assumed to be AND.

Transformation helpers subscription filters

Transformation helpers subscription filters utils list

`$util.transform.toSubscriptionFilter(Map) : Map`

Converts a `Map` input object to a `SubscriptionFilter` expression object. The `$util.transform.toSubscriptionFilter` method is used as an input to the `$extensions.setSubscriptionFilter()` extension. For more information, see Extensions.

`$util.transform.toSubscriptionFilter(Map, List) : Map`

Converts a `Map` input object to a `SubscriptionFilter` expression object. The `$util.transform.toSubscriptionFilter` method is used as an input to the `$extensions.setSubscriptionFilter()` extension. For more information, see Extensions.

The first argument is the `Map` input object that's converted to the `SubscriptionFilter` expression object. The second argument is a `List` of field names that are ignored in the first `Map` input object while constructing the `SubscriptionFilter` expression object.

`$util.transform.toSubscriptionFilter(Map, List, Map) : Map`

Converts a `Map` input object to a `SubscriptionFilter` expression object. The `$util.transform.toSubscriptionFilter` method is used as an input to the `$extensions.setSubscriptionFilter()` extension. For more information, see Extensions.

The first argument is the `Map` input object that's converted to the `SubscriptionFilter` expression object, the second argument is a `List` of field names that will be ignored in the first `Map` input object, and the third argument is a `Map` input object of strict rules that's included while constructing the `SubscriptionFilter` expression object. These strict rules are included

in the `SubscriptionFilter` expression object in such a way that at least one of the rules will be satisfied to pass the subscription filter.

Subscription filter arguments

The following table explains the how the arguments of the following utilities are defined:

- `$util.transform.toSubscriptionFilter(Map) : Map`
- `$util.transform.toSubscriptionFilter(Map, List) : Map`
- `$util.transform.toSubscriptionFilter(Map, List, Map) : Map`

Argument 1: Map

Argument 1 is a `Map` object with the following key values:

- field names
- "and"
- "or"

For field names as keys, the conditions on these fields' entries are in the form of `"operator" : "value"`.

The following example shows how entries can be added to the `Map`:

```
"field_name" : {
                    "operator1" : value
               }

## We can have multiple conditions for the same field_name:

"field_name" : {
                    "operator1" : value
                    "operator2" : value
                    .
                    .
                    .
               }
```

When a field has two or more conditions on it, all of these conditions are considered to use the OR operation.

The input `Map` can also have "and" and "or" as keys, implying that all entries within these should be joined using AND or OR logic depending on the key. The key values "and" and "or" expect an array of conditions.

```
"and" : [

            {
                "field_name1" : {
                    "operator1" : value
                }
             },

             {
                "field_name2" : {
                    "operator1" : value
                }
             },
             .
             .
        ].
```

Note that you can nest "and" and "or". That is, you can have nested "and"/"or" within another "and"/"or" block. However, this doesn't work for simple fields.

```
"and" : [

            {
                "field_name1" : {
                    "operator" : value
                }
             },

             {
                "or" : [
                            {
                                "field_name2" : {
                                    "operator" : value
                                }
                            },
```

```
                        {
                            "field_name3" : {
                                "operator" : value
                            }
                        }

                    ].
```

The following example shows an input of *argument 1* using `$util.transform.toSubscriptionFilter(Map) : Map`.

Input(s)

Argument 1: Map:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
```

```
    }
  ]
}
```

Output

The result is a Map object:

```
{
  "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

Argument 2 contains a `List` of field names that shouldn't be considered in the input `Map` (argument 1) while constructing the `SubscriptionFilter` expression object. The `List` can also be empty.

The following example shows the inputs of argument 1 and argument 2 using `$util.transform.toSubscriptionFilter(Map, List) : Map`.

Input(s)

Argument 1: Map:

```
{

  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
```

```
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argument 2: List:

```
["percentageUp", "author"]
```

Output

The result is a `Map` object:

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
```

```
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 3: Map

Argument 3 is a `Map` object that has field names as key values (cannot have "and" or "or"). For field names as keys, the conditions on these fields are entries in the form of `"operator" : "value"`. Unlike argument 1, argument 3 cannot have multiple conditions in the same key. In addition, argument 3 doesn't have an "and" or "or" clause, so there's no nesting involved either.

Argument 3 represents a list of strict rules, which are added to the `SubscriptionFilter` expression object so that **at least one** of these conditions is met to pass the filter.

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.
```

The following example shows the inputs of *argument 1*, *argument 2*, and *argument 3* using `$util.transform.toSubscriptionFilter(Map, List, Map) : Map`.

Input(s)

Argument 1: Map:

```
{
```

```
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argument 2: List:

```
["percentageUp", "author"]
```

Argument 3: Map:

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
```

```
}
```

Output

The result is a `Map` object:

```
{
  "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"
        }
      ]
    }
  ]
}
```

Math helpers in $util.math

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

`$util.math` contains methods to help with common Math operations.

$util.math utils list

`$util.math.roundNum(Double) : Integer`

Takes a double and rounds it to the nearest integer.

`$util.math.minVal(Double, Double) : Double`

Takes two doubles and returns the minimum value between the two doubles.

`$util.math.maxVal(Double, Double) : Double`

Takes two doubles and returns the maximum value between the two doubles.

`$util.math.randomDouble() : Double`

Returns a random double between 0 and 1.

> **Important**
>
> This function shouldn't be used for anything that needs high entropy randomness (for example, cryptography).

`$util.math.randomWithinRange(Integer, Integer) : Integer`

Returns a random integer value within the specified range, with the first argument specifying the lower value of the range and the second argument specifying the upper value of the range.

> **Important**
>
> This function shouldn't be used for anything that needs high entropy randomness (for example, cryptography).

String helpers in $util.str

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

`$util.str` contains methods to help with common String operations.

$util.str utils list

`$util.str.toUpper(String) : String`

Takes a string and converts it to be entirely uppercase.

`$util.str.toLower(String) : String`

Takes a string and converts it to be entirely lowercase.

`$util.str.toReplace(String, String, String) : String`

Replaces a substring within a string with another string. The first argument specifies the string on which to perform the replacement operation. The second argument specifies the substring

to replace. The third argument specifies the string to replace the second argument with. The following is an example of this utility's usage:

```
INPUT:      $util.str.toReplace("hello world", "hello", "mellow")
OUTPUT:     "mellow world"
```

`$util.str.normalize(String, String) : String`

Normalizes a string using one of the four unicode normalization forms: NFC, NFD, NFKC, or NFKD. The first argument is the string to normalize. The second argument is either "nfc", "nfd", "nfkc", or "nfkd" specifying the normalization type to use for the normalization process.

Extensions

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

`$extensions` contains a set of methods to make additional actions within your resolvers.

Caching extensions

`$extensions.evictFromApiCache(String, String, Object) : Object`

Evicts an item from the AWS AppSync server-side cache. The first argument is the type name. The second argument is the field name. The third argument is an object containing key-value pair items that specify the caching key value. You must put the items in the object in the same order as the caching keys in the cached resolver's `cachingKey`.

> **Note**
>
> This utility works only for mutations, not queries.

Subscription extensions

`$extensions.setSubscriptionFilter(filterJsonObject)`

Defines enhanced subscription filters. Each subscription notification event is evaluated against provided subscription filters and delivers notifications to clients if all filters evaluate to `true`. The argument is `filterJsonObject` as described in the following section.

> **Note**
>
> You can use this extension method only in the response mapping templates of a subscription resolver.

`$extensions.setSubscriptionInvalidationFilter(filterJsonObject)`

Defines subscription invalidation filters. Subscription filters are evaluated against the invalidation payload, then invalidate a given subscription if the filters evaluate to `true`. The argument is `filterJsonObject` as described in the following section.

> **Note**
>
> You can use this extension method only in the response mapping templates of a subscription resolver.

`$extensions.invalidateSubscriptions(invalidationJsonObject)`

Used to initiate a subscription invalidation from a mutation. The argument is `invalidationJsonObject` as described in the following section.

> **Note**
>
> This extension can be used only in the response mapping templates of the mutation resolvers.
> You can only use at most five unique `$extensions.invalidateSubscriptions()` method calls in any single request. If you exceed this limit, you will receive a GraphQL error.

Argument: filterJsonObject

The JSON object defines either subscription or invalidation filters. It's an array of filters in a `filterGroup`. Each filter is a collection of individual filters.

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

Each filter has three attributes:

- `fieldName` – The GraphQL schema field.
- `operator` – The operator type.
- `value` – The values to compare to the subscription notification `fieldName` value.

The following is an example assignment of these attributes:

```
{
 "fieldName" : "severity",
 "operator" : "le",
```

```
 "value" : $context.result.severity
}
```

Field: fieldName

The string type `fieldName` refers to a field defined in the GraphQL schema that matches the `fieldName` in the subscription notification payload. When a match is found, the `value` of the GraphQL schema field is compared to the `value` of the subscription notification filter. In the following example, the `fieldName` filter matches the `service` field defined in a given GraphQL type. If the notification payload contains a `service` field with a `value` equivalent to AWS AppSync, the filter evaluates to `true`:

```
{
 "fieldName" : "service",
 "operator" : "eq",
 "value" : "AWS AppSync"
}
```

Field: value

The value can be a different type based on the operator:

- A single number or Boolean
  - String examples: `"test"`, `"service"`
  - Number examples: 1, 2, `45.75`
  - Boolean examples: `true`, `false`
- Pairs of numbers or strings
  - String pair example: `["test1","test2"]`, `["start","end"]`
  - Number pair example: `[1,4]`, `[67,89]`, `[12.45, 95.45]`
- Arrays of numbers or strings
  - String array example: `["test1","test2","test3","test4","test5"]`
  - Number array example: `[1,2,3,4,5]`, `[12.11,46.13,45.09,12.54,13.89]`

Field: operator

A case-sensitive string with the following possible values:

Operator	Description	Possible value types
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

The following table describes how each operator is used in the subscription notification.

eq (equal)

The `eq` operator evaluates to `true` if the subscription notification field value matches and is strictly equal to the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `service` field with the value equivalent to `AWS AppSync`.

Possible value types: integer, float, string, Boolean

```
{
 "fieldName" : "service",
 "operator" : "eq",
 "value" : "AWS AppSync"
}
```

ne (not equal)

The ne operator evaluates to `true` if the subscription notification field value is different from the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `service` field with a value different from `AWS AppSync`.

Possible value types: integer, float, string, Boolean

```
{
 "fieldName" : "service",
 "operator" : "ne",
 "value" : "AWS AppSync"
}
```

le (less or equal)

The `le` operator evaluates to `true` if the subscription notification field value is less than or equal to the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `size` field with a value less than or equal to 5.

Possible value types: integer, float, string

```
{
 "fieldName" : "size",
 "operator" : "le",
 "value" : 5
}
```

lt (less than)

The `lt` operator evaluates to `true` if the subscription notification field value is lower than the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `size` field with a value lower than 5.

Possible value types: integer, float, string

```
{
 "fieldName" : "size",
 "operator" : "lt",
 "value" : 5
}
```

ge (greater or equal)

The ge operator evaluates to `true` if the subscription notification field value is greater than or equal to the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `size`field with a value greater than or equal to 5.

Possible value types: integer, float, string

```
{
 "fieldName" : "size",
 "operator" : "ge",
 "value" : 5
}
```

gt (greater than)

The `gt` operator evaluates to `true` if the subscription notification field value is greater than the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `size` field with a value greater than 5.

Possible value types: integer, float, string

```
{
 "fieldName" : "size",
 "operator" : "gt",
 "value" : 5
}
```

contains

The `contains` operator checks for a substring, subsequence, or value in a set or single item. A filter with the `contains` operator evaluates to `true` if the subscription notification field value contains the filter value. In the following example, the filter evaluates to `true` if the subscription notification has a `seats` field with the array value containing the value `10`.

Possible value types: integer, float, string

```
{
 "fieldName" : "seats",
 "operator" : "contains",
 "value" : 10
}
```

In another example, the filter evaluates to `true` if the subscription notification has an `event` field with `launch` as substring.

```
{
 "fieldName" : "event",
 "operator" : "contains",
 "value" : "launch"
}
```

notContains

The `notContains` operator checks for the absence of a substring, subsequence, or value in a set or single item. The filter with the `notContains` operator evaluates to `true` if the subscription notification field value doesn't contain the filter value. In the following example, the filter evaluates to `true` if the subscription notification has a `seats` field with the array value not containing the value `10`.

Possible value types: integer, float, string

```
{
 "fieldName" : "seats",
 "operator" : "notContains",
 "value" : 10
}
```

In another example, filter evaluates to `true` if the subscription notification has an `event` field value without `launch` as its subsequence.

```
{
 "fieldName" : "event",
 "operator" : "notContains",
 "value" : "launch"
}
```

beginsWith

The `beginsWith` operator checks for a prefix in a string. The filter containing the `beginsWith` operator evaluates to `true` if the subscription notification field value begins with the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `service` field with a value that begins with `AWS`.

Possible value type: string

```
{
 "fieldName" : "service",
 "operator" : "beginsWith",
 "value" : "AWS"
}
```

in

The `in` operator checks for matching elements in an array. The filter containing the `in` operator evaluates to `true` if the subscription notification field value exists in an array. In the following example, the filter evaluates to `true` if the subscription notification has a `severity` field with one of the values present in the array: `[1,2,3]`.

Possible value type: Array of integer, float, or string

```
{
 "fieldName" : "severity",
 "operator" : "in",
 "value" : [1,2,3]
}
```

notIn

The `notIn` operator checks for missing elements in an array. The filter containing the `notIn` operator evaluates to `true` if the subscription notification field value doesn't exist in the array. In the following example, the filter evaluates to `true` if the subscription notification has a `severity` field with one of the values not present in the array: `[1,2,3]`.

Possible value type: Array of integer, float, or string

```
{
 "fieldName" : "severity",
```

```
 "operator" : "notIn",
 "value" : [1,2,3]
}
```

between

The `between` operator checks for values between two numbers or strings. The filter containing the `between` operator evaluates to `true` if the subscription notification field value is between the filter's value pair. In the following example, the filter evaluates to `true` if the subscription notification has a `severity` field with values 2,3,4.

Possible value types: Pair of integer, float, or string

```
{
 "fieldName" : "severity",
 "operator" : "between",
 "value" : [1,5]
}
```

containsAny

The `containsAny` operator checks for common elements in arrays. A filter with the `containsAny` operator evaluates to `true` if the intersection of the subscription notification field set value and filter set value is non empty. In the following example, the filter evaluates to `true` if the subscription notification has a `seats` field with an array value containing either `10` or `15`. This means that filter would evaluate to `true` if the subscription notification had a `seats` field value of `[10,11]` or `[15,20,30]`.

Possible value types: integer, float, or string

```
{
 "fieldName" : "seats",
 "operator" : "containsAny",
 "value" : [10, 15]
}
```

AND logic

You can combine multiple filters using AND logic by defining multiple entries within the `filters` object in the `filterGroup` array. In the following example, filters evaluate to `true` if the

subscription notification has a `userId` field with a value equivalent to 1 AND a `group` field value of either `Admin` or `Developer`.

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

OR logic

You can combine multiple filters using OR logic by defining multiple filter objects within the `filterGroup` array. In the following example, filters evaluate to `true` if the subscription notification has a `userId` field with a value equivalent to 1 OR a `group` field value of either `Admin` or `Developer`.

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
```

```
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

Exceptions

Note that there are several restrictions for using filters:

- In the `filters` object, there can be a maximum of five unique `fieldName` items per filter. This means that you can combine a maximum of five individual `fieldName` objects using AND logic.
- There can be a maximum of twenty values for the `containsAny` operator.
- There can be a maximum of five values for the `in` and `notIn` operators.
- Each string can be a maximum of 256 characters.
- Each string comparison is case sensitive.
- Nested object filtering allows up to five nested levels of filtering.
- Each `filterGroup` can have a maximum of 10 `filters`. This means that you can combine a maximum of 10 `filters` using OR logic.
  - The `in` operator is a special case of OR logic. In the following example, there are two `filters`:

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
```

```
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
            ]
        }
    ]
}
```

The preceding filter group is evaluated as follows and counts towards the maximum filters limit:

```
{
    "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"
                }
           ]
        }
    ]
```

```
}
```

Argument: invalidationJsonObject

The `invalidationJsonObject` defines the following:

- `subscriptionField` – The GraphQL schema subscription to invalidate. A single subscription, defined as a string in the `subscriptionField`, is considered for invalidation.
- `payload` – A key-value pair list that's used as the input for invalidating subscriptions if the invalidation filter evaluates to `true` against their values.

  The following example invalidates subscribed and connected clients using the `onUserDelete` subscription when the invalidation filter defined in the subscription resolver evaluates to `true` against the `payload` value.

```
$extensions.invalidateSubscriptions({
        "subscriptionField": "onUserDelete",
        "payload": {
                "group": "Developer"
                "type" : "Full-Time"
      }
    })
```

AWS AppSync resolver mapping template reference for DynamoDB

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

The AWS AppSync DynamoDB function allows you to use GraphQL to store and retrieve data in existing Amazon DynamoDB tables in your account by mapping an incoming GraphQL request into a DynamoDB call, and then mapping the DynamoDB response back to GraphQL. This section describes the request and response handlers for supported DynamoDB operations:

- GetItem - The GetItem request lets you tell the DynamoDB function to make a GetItem request to DynamoDB, and enables you to specify the key of the item in DynamoDB and whether to use a consistent read or not.
- PutItem - The PutItem request mapping document lets you tell the DynamoDB function to make a PutItem request to DynamoDB, and enables you to specify the key of the item in DynamoDB, the full contents of the item (composed of key and attributeValues), and conditions for the operation to succeed.
- UpdateItem - The UpdateItem request enables you to tell the DynamoDB function to make a UpdateItem request to DynamoDB and allows you to specify the key of the item in DynamoDB, an update expression describing how to update the item in DynamoDB, and conditions for the operation to succeed.
- DeleteItem - The DeleteItem request lets you tell the DynamoDB function to make a DeleteItem request to DynamoDB, and enables you to specify the key of the item in DynamoDB and conditions for the operation to succeed.
- Query - The Query request object lets you tell the DynamoDB resolver to make a Query request to DynamoDB, and enables you to specify the key expression, which index to use, additional filters, how many items to return, whether to use consistent reads, query direction (forward or backward), and pagination tokens.
- Scan - The Scan request lets you tell the DynamoDB function to make a Scan request to DynamoDB, and enables you to specify a filter to exclude results, which index to use, how many items to return, whether to use consistent reads, pagination tokens, and parallel scans.
- Sync - The Sync request object lets you retrieve all the results from a DynamoDB table and then receive only the data altered since your last query (the delta updates). Sync requests can only be made to versioned DynamoDB data sources. You can specify a filter to exclude results, how many items to return, pagination Tokens, and when your last Sync operation was started.
- BatchGetItem - The BatchGetItem request object lets you tell the DynamoDB function to make a BatchGetItem request to DynamoDB to retrieve multiple items, potentially across multiple tables. For this request object, you must specify the table names to retrieve the items from and the keys of the items to retrieve from each table.
- BatchDeleteItem - The BatchDeleteItem request object lets you tell the DynamoDB function to make a BatchWriteItem request to DynamoDB to delete multiple items, potentially across multiple tables. For this request object, you must specify the table names to delete the items from and the keys of the items to delete from each table.
- BatchPutItem - The BatchPutItem request object lets you tell the DynamoDB function to make a BatchWriteItem request to DynamoDB to put multiple items, potentially across multiple tables.

For this request object, you must specify the table names to put the items in and the full items to put in each table.

- TransactGetItems - The TransactGetItems request object lets you to tell the DynamoDB function to make a TransactGetItems request to DynamoDB to retrieve multiple items, potentially across multiple tables. For this request object, you must specify the table name of each request item to retrieve the item from and the key of each request item to retrieve from each table.
- TransactWriteItems - The TransactWriteItems request object lets you tell the DynamoDB function to make a TransactWriteItems request to DynamoDB to write multiple items, potentially to multiple tables. For this request object, you must specify the destination table name of each request item, the operation of each request item to perform, and the key of each request item to write.
- Type system (request mapping) - Learn more about how DynamoDB typing is integrated into AWS AppSync requests.
- Type system (response mapping) - Learn more about how DynamoDB types are converted automatically to GraphQL or JSON in a response payload.
- Filters - Learn more about filters for query and scan operations.
- Condition expressions - Learn more about condition expressions for PutItem, UpdateItem, and DeleteItem operations.
- Transaction condition expressions - Learn more about condition expressions for TransactWriteItems operations.
- Projections - Learn more about how to specify attributes in read operations.

GetItem

The `GetItem` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `GetItem` request to DynamoDB, and enables you to specify:

- The key of the item in DynamoDB
- Whether to use a consistent read or not

The `GetItem` mapping document has the following structure:

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
```

```
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true,
    "projection" : {
        ...
    }
}
```

The fields are defined as follows:

GetItem fields

GetItem fields list

`version`

The template definition version. `2017-02-28` and `2018-05-29` are currently supported. This value is required.

`operation`

The DynamoDB operation to perform. To perform the `GetItem` DynamoDB operation, this must be set to `GetItem`. This value is required.

`key`

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a "typed value", see Type system (request mapping). This value is required.

`consistentRead`

Whether or not to perform a strongly consistent read with DynamoDB. This is optional, and defaults to `false`.

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

The item returned from DynamoDB is automatically converted into GraphQL and JSON primitive types, and is available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about response mapping templates, see Resolver mapping template overview.

Example

The following example is a mapping template for a GraphQL query `getThing(foo: String!, bar: String!)`:

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
        "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
    },
    "consistentRead" : true
}
```

For more information about the DynamoDB `GetItem` API, see the DynamoDB API documentation.

PutItem

The `PutItem` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `PutItem` request to DynamoDB, and enables you to specify the following:

- The key of the item in DynamoDB
- The full contents of the item (composed of `key` and `attributeValues`)
- Conditions for the operation to succeed

The `PutItem` mapping document has the following structure:

```
{
    "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
}
```

The fields are defined as follows:

PutItem fields

PutItem fields list

`version`

The template definition version. `2017-02-28` and `2018-05-29` are currently supported. This value is required.

`operation`

The DynamoDB operation to perform. To perform the `PutItem` DynamoDB operation, this must be set to `PutItem`. This value is required.

`key`

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a "typed value", see Type system (request mapping). This value is required.

`attributeValues`

The rest of the attributes of the item to be put into DynamoDB. For more information about how to specify a "typed value", see Type system (request mapping). This field is optional.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `PutItem` request overwrites any existing entry for that item. For more information about conditions, see Condition expressions. This value is optional.

`_version`

A numeric value that represents the latest known version of an item. This value is optional. This field is used for *Conflict Detection* and is only supported on versioned data sources.

`customPartitionKey`

When enabled, this string value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled (for more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*). When enabled, the processing of the `populateIndexFields` entry is also enabled. This field is optional.

`populateIndexFields`

A boolean value that, when enabled **along with the `customPartitionKey`**, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns. For more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*. This field is optional.

The item written to DynamoDB is automatically converted into GraphQL and JSON primitive types and is available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about response mapping templates, see Resolver mapping template overview.

Example 1

The following example is a mapping template for a GraphQL mutation `updateThing(foo: String!, bar: String!, name: String!, version: Int!)`.

If no item with the specified key exists, it's created. If an item already exists with the specified key, it's overwritten.

```
{
    "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)
    }
}
```

Example 2

The following example is a mapping template for a GraphQL mutation `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)`.

This example checks to be sure the item currently in DynamoDB has the `version` field set to `expectedVersion`.

```
{
    "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)
        }
    }
}
```

For more information about the DynamoDB `PutItem` API, see the DynamoDB API documentation.

UpdateItem

The `UpdateItem` request mapping document enables you to tell the AWS AppSync DynamoDB resolver to make a `UpdateItem` request to DynamoDB and allows you to specify the following:

- The key of the item in DynamoDB
- An update expression describing how to update the item in DynamoDB

- Conditions for the operation to succeed

The `UpdateItem` mapping document has the following structure:

```
{
    "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
}
```

The fields are defined as follows:

UpdateItem fields

UpdateItem fields list

`version`

The template definition version. `2017-02-28` and `2018-05-29` are currently supported. This value is required.

`operation`

The DynamoDB operation to perform. To perform the `UpdateItem` DynamoDB operation, this must be set to `UpdateItem`. This value is required.

key

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about specifying a "typed value", see Type system (request mapping). This value is required.

update

The `update` section lets you specify an update expression that describes how to update the item in DynamoDB. For more information about how to write update expressions, see the DynamoDB UpdateExpressions documentation. This section is required.

The `update` section has three components:

expression

The update expression. This value is required.

expressionNames

The substitutions for expression attribute *name* placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the `expression`, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the `expression`.

expressionValues

The substitutions for expression attribute *value* placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the `expression`, and the value must be a typed value. For more information about how to specify a "typed value", see Type system (request mapping). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the `expression`.

condition

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `UpdateItem` request updates the existing entry regardless of its current state. For more information about conditions, see Condition expressions. This value is optional.

`_version`

A numeric value that represents the latest known version of an item. This value is optional. This field is used for *Conflict Detection* and is only supported on versioned data sources.

`customPartitionKey`

When enabled, this string value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled (for more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*). When enabled, the processing of the `populateIndexFields` entry is also enabled. This field is optional.

`populateIndexFields`

A boolean value that, when enabled **along with the `customPartitionKey`**, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns. For more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*. This field is optional.

The item updated in DynamoDB is automatically converted into GraphQL and JSON primitive types and is available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about response mapping templates, see Resolver mapping template overview.

Example 1

The following example is a mapping template for the GraphQL mutation `upvote(id: ID!)`.

In this example, an item in DynamoDB has its `upvotes` and `version` fields incremented by 1.

```
{
    "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 }
        }
    }
}
```

Example 2

The following example is a mapping template for a GraphQL mutation `updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)`.

This is a complex example that inspects the arguments and dynamically generates the update expression that only includes the arguments that have been provided by the client. For example, if `title` and `author` are omitted, they are not updated. If an argument is specified but its value is `null`, then that field is deleted from the object in DynamoDB. Finally, the operation has a condition, which verifies whether the item currently in DynamoDB has the `version` field set to `expectedVersion`:

```
{
    "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)
        }
    }
}
```

For more information about the DynamoDB `UpdateItem` API, see the DynamoDB API documentation.

DeleteItem

The `DeleteItem` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `DeleteItem` request to DynamoDB, and enables you to specify the following:

- The key of the item in DynamoDB
- Conditions for the operation to succeed

The `DeleteItem` mapping document has the following structure:

```
{
    "version" : "2018-05-29",
    "operation" : "DeleteItem",
    "customPartitionKey" : "foo",
    "populateIndexFields" : boolean value,
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "condition" : {
        ...
    },
    "_version" : 1
}
```

The fields are defined as follows:

DeleteItem fields

DeleteItem fields list

`version`

The template definition version. `2017-02-28` and `2018-05-29` are currently supported. This value is required.

`operation`

The DynamoDB operation to perform. To perform the `DeleteItem` DynamoDB operation, this must be set to `DeleteItem`. This value is required.

`key`

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about specifying a “typed value”, see Type system (request mapping). This value is required.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `DeleteItem` request deletes an item regardless of its current state. For more information about conditions, see Condition expressions. This value is optional.

`_version`

A numeric value that represents the latest known version of an item. This value is optional. This field is used for *Conflict Detection* and is only supported on versioned data sources.

`customPartitionKey`

When enabled, this string value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled (for more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*). When enabled, the processing of the `populateIndexFields` entry is also enabled. This field is optional.

`populateIndexFields`

A boolean value that, when enabled **along with the `customPartitionKey`**, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns. For more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*. This field is optional.

The item deleted from DynamoDB is automatically converted into GraphQL and JSON primitive types and is available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about response mapping templates, see Resolver mapping template overview.

Example 1

The following example is a mapping template for a GraphQL mutation `deleteItem(id: ID!)`. If an item exists with this ID, it's deleted.

```
{
    "version" : "2017-02-28",
```

```
    "operation" : "DeleteItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

Example 2

The following example is a mapping template for a GraphQL mutation `deleteItem(id: ID!, expectedVersion: Int!)`. If an item exists with this ID, it's deleted, but only if its `version` field set to `expectedVersion`:

```
{
    "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)
        }
    }
}
```

For more information about the DynamoDB `DeleteItem` API, see the DynamoDB API documentation.

Query

The `Query` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `Query` request to DynamoDB, and enables you to specify the following:

- Key expression
- Which index to use
- Any additional filter
- How many items to return
- Whether to use consistent reads

- query direction (forward or backward)
- Pagination token

The `Query` mapping document has the following structure:

```
{
    "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" : {
        ...
    }
}
```

The fields are defined as follows:

Query fields

Query fields list

`version`

The template definition version. `2017-02-28` and `2018-05-29` are currently supported. This value is required.

`operation`

The DynamoDB operation to perform. To perform the `Query` DynamoDB operation, this must be set to `Query`. This value is required.

`query`

The `query` section lets you specify a key condition expression that describes which items to retrieve from DynamoDB. For more information about how to write key condition expressions, see the DynamoDB KeyConditions documentation . This section must be specified.

`expression`

The query expression. This field must be specified.

`expressionNames`

The substitutions for expression attribute *name* placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the `expression`, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the `expression`.

`expressionValues`

The substitutions for expression attribute *value* placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the `expression`, and the value must be a typed value. For more information about how to specify a “typed value”, see Type system (request mapping). This value is required. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the `expression`.

`filter`

An additional filter that can be used to filter the results from DynamoDB before they are returned. For more information about filters, see Filters. This field is optional.

`index`

The name of the index to query. The DynamoDB query operation allows you to scan on Local Secondary Indexes and Global Secondary Indexes in addition to the primary key index for a hash key. If specified, this tells DynamoDB to query the specified index. If omitted, the primary key index is queried.

`nextToken`

The pagination token to continue a previous query. This would have been obtained from a previous query. This field is optional.

`limit`

The maximum number of items to evaluate (not necessarily the number of matching items). This field is optional.

`scanIndexForward`

A boolean indicating whether to query forwards or backwards. This field is optional, and defaults to `true`.

`consistentRead`

A boolean indicating whether to use consistent reads when querying DynamoDB. This field is optional, and defaults to `false`.

`select`

By default, the AWS AppSync DynamoDB resolver only returns attributes that are projected into the index. If more attributes are required, you can set this field. This field is optional. The supported values are:

`ALL_ATTRIBUTES`

Returns all of the item attributes from the specified table or index. If you query a local secondary index, DynamoDB fetches the entire item from the parent table for each matching item in the index. If the index is configured to project all item attributes, all of the data can be obtained from the local secondary index and no fetching is required.

`ALL_PROJECTED_ATTRIBUTES`

Allowed only when querying an index. Retrieves all attributes that have been projected into the index. If the index is configured to project all attributes, this return value is equivalent to specifying `ALL_ATTRIBUTES`.

`SPECIFIC_ATTRIBUTES`

Returns only the attributes listed in the `projection`'s `expression`. This return value is equivalent to specifying the `projection`'s `expression` without specifying any value for `Select`.

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

The results from DynamoDB are automatically converted into GraphQL and JSON primitive types and are available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about response mapping templates, see Resolver mapping template overview.

The results have the following structure:

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

The fields are defined as follows:

`items`

A list containing the items returned by the DynamoDB query.

`nextToken`

If there might be more results, `nextToken` contains a pagination token that you can use in another request. Note that AWS AppSync encrypts and obfuscates the pagination token returned from DynamoDB. This prevents your table data from being inadvertently leaked to the caller. Also note that these pagination tokens cannot be used across different resolvers.

`scannedCount`

The number of items that matched the query condition expression, before a filter expression (if present) was applied.

Example

The following example is a mapping template for a GraphQL query `getPosts(owner: ID!)`.

In this example, a global secondary index on a table is queried to return all posts owned by the specified ID.

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "query" : {
        "expression" : "ownerId = :ownerId",
        "expressionValues" : {
            ":ownerId" : $util.dynamodb.toDynamoDBJson($context.arguments.owner)
        }
    },
    "index" : "owner-index"
}
```

For more information about the DynamoDB `Query` API, see the DynamoDB API documentation.

Scan

The `Scan` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `Scan` request to DynamoDB, and enables you to specify the following:

- A filter to exclude results
- Which index to use
- How many items to return
- Whether to use consistent reads
- Pagination token
- Parallel scans

The `Scan` mapping document has the following structure:

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "index" : "fooIndex",
    "limit" : 10,
    "consistentRead" : false,
    "nextToken" : "aPaginationToken",
    "totalSegments" : 10,
```

```
    "segment" : 1,
    "filter" : {
        ...
    },
    "projection" : {
        ...
    }
}
```

The fields are defined as follows:

Scan fields

Scan fields list

`version`

The template definition version. `2017-02-28` and `2018-05-29` are currently supported. This value is required.

`operation`

The DynamoDB operation to perform. To perform the `Scan` DynamoDB operation, this must be set to `Scan`. This value is required.

`filter`

A filter that can be used to filter the results from DynamoDB before they are returned. For more information about filters, see Filters. This field is optional.

`index`

The name of the index to query. The DynamoDB query operation allows you to scan on Local Secondary Indexes and Global Secondary Indexes in addition to the primary key index for a hash key. If specified, this tells DynamoDB to query the specified index. If omitted, the primary key index is queried.

`limit`

The maximum number of items to evaluate at a single time. This field is optional.

`consistentRead`

A Boolean that indicates whether to use consistent reads when querying DynamoDB. This field is optional, and defaults to `false`.

`nextToken`

The pagination token to continue a previous query. This would have been obtained from a previous query. This field is optional.

`select`

By default, the AWS AppSync DynamoDB resolver only returns whatever attributes are projected into the index. If more attributes are required, then this field can be set. This field is optional. The supported values are:

`ALL_ATTRIBUTES`

Returns all of the item attributes from the specified table or index. If you query a local secondary index, DynamoDB fetches the entire item from the parent table for each matching item in the index. If the index is configured to project all item attributes, all of the data can be obtained from the local secondary index and no fetching is required.

`ALL_PROJECTED_ATTRIBUTES`

Allowed only when querying an index. Retrieves all attributes that have been projected into the index. If the index is configured to project all attributes, this return value is equivalent to specifying `ALL_ATTRIBUTES`.

`SPECIFIC_ATTRIBUTES`

Returns only the attributes listed in the `projection`'s `expression`. This return value is equivalent to specifying the `projection`'s `expression` without specifying any value for `Select`.

`totalSegments`

The number of segments to partition the table by when performing a parallel scan. This field is optional, but must be specified if `segment` is specified.

`segment`

The table segment in this operation when performing a parallel scan. This field is optional, but must be specified if `totalSegments` is specified.

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

The results returned by the DynamoDB scan are automatically converted into GraphQL and JSON primitive types and is available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about response mapping templates, see Resolver mapping template overview.

The results have the following structure:

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

The fields are defined as follows:

`items`

A list containing the items returned by the DynamoDB scan.

`nextToken`

If there might be more results, `nextToken` contains a pagination token that you can use in another request. AWS AppSync encrypts and obfuscates the pagination token returned from DynamoDB. This prevents your table data from being inadvertently leaked to the caller. Also, these pagination tokens can't be used across different resolvers.

`scannedCount`

The number of items that were retrieved by DynamoDB before a filter expression (if present) was applied.

Example 1

The following example is a mapping template for the GraphQL query: `allPosts`.

In this example, all entries in the table are returned.

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

```
}
```

Example 2

The following example is a mapping template for the GraphQL query: `postsMatching(title: String!)`.

In this example, all entries in the table are returned where the title starts with the `title` argument.

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "filter" : {
        "expression" : "begins_with(title, :title)",
        "expressionValues" : {
            ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
        },
    }
}
```

For more information about the DynamoDB `Scan` API, see the DynamoDB API documentation.

Sync

The `Sync` request mapping document lets you retrieve all the results from a DynamoDB table and then receive only the data altered since your last query (the delta updates). `Sync` requests can only be made to versioned DynamoDB data sources. You can specify the following:

- A filter to exclude results
- How many items to return
- Pagination Token
- When your last `Sync` operation was started

The `Sync` mapping document has the following structure:

```
{
    "version" : "2018-05-29",
    "operation" : "Sync",
```

```
    "basePartitionKey": "Base Tables PartitionKey",
    "deltaIndexName": "delta-index-name",
    "limit" : 10,
    "nextToken" : "aPaginationToken",
    "lastSync" :  1550000000000,
    "filter" : {
        ...
    }
}
```

The fields are defined as follows:

Sync fields

Sync fields list

`version`

The template definition version. Only `2018-05-29` is currently supported. This value is required.

`operation`

The DynamoDB operation to perform. To perform the `Sync` operation, this must be set to `Sync`. This value is required.

`filter`

A filter that can be used to filter the results from DynamoDB before they are returned. For more information about filters, see Filters. This field is optional.

`limit`

The maximum number of items to evaluate at a single time. This field is optional. If omitted, the default limit will be set to `100` items. The maximum value for this field is `1000` items.

`nextToken`

The pagination token to continue a previous query. This would have been obtained from a previous query. This field is optional.

`lastSync`

The moment, in epoch milliseconds, when the last successful `Sync` operation started. If specified, only items that have changed after `lastSync` are returned. This field is optional, and

should only be populated after retrieving all pages from an initial Sync operation. If omitted, results from the *Base* table will be returned, otherwise, results from the *Delta* table will be returned.

basePartitionKey

The partition key of the *Base* table used when performing a Sync operation. This field allows a Sync operation to be performed when the table utilizes a custom partition key. This is an optional field.

deltaIndexName

The index used for the Sync operation. This index is required to enable a Sync operation on the whole delta store table when the table uses a custom partition key. The Sync operation will be performed on the GSI (created on `gsi_ds_pk` and `gsi_ds_sk`). This field is optional.

The results returned by the DynamoDB sync are automatically converted into GraphQL and JSON primitive types and are available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about response mapping templates, see Resolver mapping template overview.

The results have the following structure:

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10,
    startedAt = 1550000000000
}
```

The fields are defined as follows:

items

A list containing the items returned by the sync.

nextToken

If there might be more results, `nextToken` contains a pagination token that you can use in another request. AWS AppSync encrypts and obfuscates the pagination token returned from

DynamoDB. This prevents your table data from being inadvertently leaked to the caller. Also, these pagination tokens can't be used across different resolvers.

`scannedCount`

The number of items that were retrieved by DynamoDB before a filter expression (if present) was applied.

`startedAt`

The moment, in epoch milliseconds, when the sync operation started that you can store locally and use in another request as your `lastSync` argument. If a pagination token was included in the request, this value will be the same as the one returned by the request for the first page of results.

Example

The following example is a mapping template for the GraphQL query: `syncPosts(nextToken: String, lastSync: AWSTimestamp)`.

In this example, if `lastSync` is omitted, all entries in the base table are returned. If `lastSync` is supplied, only the entries in the delta sync table that have changed since `lastSync` are returned.

```
{
    "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

The `BatchGetItem` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `BatchGetItem` request to DynamoDB to retrieve multiple items, potentially across multiple tables. For this request template, you must specify the following:

- The table names where to retrieve the items from
- The keys of the items to retrieve from each table

The DynamoDB `BatchGetItem` limits apply and **no condition expression** can be provided.

The `BatchGetItem` mapping document has the following structure:

```
{
    "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" : {
                ...
            }
        }
    }
```

```
}
```

The fields are defined as follows:

BatchGetItem fields

BatchGetItem fields list

`version`

The template definition version. Only `2018-05-29` is supported. This value is required.

`operation`

The DynamoDB operation to perform. To perform the `BatchGetItem` DynamoDB operation, this must be set to `BatchGetItem`. This value is required.

`tables`

The DynamoDB tables to retrieve the items from. The value is a map where table names are specified as the keys of the map. At least one table must be provided. This `tables` value is required.

`keys`

List of DynamoDB keys representing the primary key of the items to retrieve. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping).

`consistentRead`

Whether to use a consistent read when executing a *GetItem* operation. This value is optional and defaults to *false*.

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

Things to remember:

- If an item has not been retrieved from the table, a *null* element appears in the data block for that table.

- Invocation results are sorted per table, based on the order in which they were provided inside the request mapping template.
- Each `Get` command inside a `BatchGetItem` is atomic, however, a batch can be partially processed. If a batch is partially processed due to an error, the unprocessed keys are returned as part of the invocation result inside the *unprocessedKeys* block.
- `BatchGetItem` is limited to 100 keys.

For the following example request mapping template:

```
{
  "version": "2018-05-29",
  "operation": "BatchGetItem",
  "tables": {
    "authors": [
        {
          "author_id": {
            "S": "a1"
          }
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          }
        }
    ],
  }
}
```

The invocation result available in `$ctx.result` is as follows:

```
{
   "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": []
    }
}
```

The `$ctx.error` contains details about the error. The keys **data**, **unprocessedKeys**, and each table key that was provided in the request mapping template are guaranteed to be present in the invocation result. Items that have been deleted appear in the **data** block. Items that haven't been processed are marked as *null* inside the data block and are placed inside the **unprocessedKeys** block.

For a more complete example, follow the DynamoDB Batch tutorial with AppSync here Tutorial: DynamoDB batch resolvers.

BatchDeleteItem

The `BatchDeleteItem` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `BatchWriteItem` request to DynamoDB to delete multiple items, potentially across multiple tables. For this request template, you must specify the following:

- The table names where to delete the items from
- The keys of the items to delete from each table

The DynamoDB `BatchWriteItem` limits apply and **no condition expression** can be provided.

The `BatchDeleteItem` mapping document has the following structure:

```
{
```

```
    "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
        }],
    }
}
```

The fields are defined as follows:

BatchDeleteItem fields

BatchDeleteItem fields list

`version`

The template definition version. Only `2018-05-29` is supported. This value is required.

`operation`

The DynamoDB operation to perform. To perform the `BatchDeleteItem` DynamoDB operation, this must be set to `BatchDeleteItem`. This value is required.

`tables`

The DynamoDB tables to delete the items from. Each table is a list of DynamoDB keys representing the primary key of the items to delete. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a "typed value", see Type system (request mapping). At least one table must be provided. The `tables` value is required.

Things to remember:

- Contrary to the `DeleteItem` operation, the fully deleted item isn't returned in the response. Only the passed key is returned.
- If an item has not been deleted from the table, a *null* element appears in the data block for that table.
- Invocation results are sorted per table, based on the order in which they were provided inside the request mapping template.
- Each `Delete` command inside a `BatchDeleteItem` is atomic. However a batch can be partially processed. If a batch is partially processed due to an error, the unprocessed keys are returned as part of the invocation result inside the *unprocessedKeys* block.
- `BatchDeleteItem` is limited to 25 keys.
- This operation **is not** supported when used with conflict detection. Using both at the same time may result in an error.

For the following example request mapping template:

```
{
  "version": "2018-05-29",
  "operation": "BatchDeleteItem",
  "tables": {
    "authors": [
        {
          "author_id": {
            "S": "a1"
          }
        },
    ],
    "posts": [
        {
```

```
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          }
        }
    ],
  }
}
```

The invocation result available in `$ctx.result` is as follows:

```
{
   "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": []
   }
}
```

The `$ctx.error` contains details about the error. The keys **data**, **unprocessedKeys**, and each table key that was provided in the request mapping template are guaranteed to be present in the invocation result. Items that have been deleted are present in the **data** block. Items that haven't been processed are marked as *null* inside the data block and are placed inside the **unprocessedKeys** block.

For a more complete example, follow the DynamoDB Batch tutorial with AppSync here Tutorial: DynamoDB batch resolvers.

BatchPutItem

The `BatchPutItem` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `BatchWriteItem` request to DynamoDB to put multiple items, potentially across multiple tables. For this request template, you must specify the following:

- The table names where to put the items in
- The full items to put in each table

The DynamoDB `BatchWriteItem` limits apply and **no condition expression** can be provided.

The `BatchPutItem` mapping document has the following structure:

```
{
    "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
        }],
```

```
    }
}
```

The fields are defined as follows:

BatchPutItem fields

BatchPutItem fields list

`version`

The template definition version. Only `2018-05-29` is supported. This value is required.

`operation`

The DynamoDB operation to perform. To perform the `BatchPutItem` DynamoDB operation, this must be set to `BatchPutItem`. This value is required.

`tables`

The DynamoDB tables to put the items in. Each table entry represents a list of DynamoDB items to insert for this specific table. At least one table must be provided. This value is required.

Things to remember:

- The fully inserted items are returned in the response, if successful.
- If an item hasn't been inserted in the table, a *null* element is displayed in the data block for that table.
- The inserted items are sorted per table, based on the order in which they were provided inside the request mapping template.
- Each `Put` command inside a `BatchPutItem` is atomic, however, a batch can be partially processed. If a batch is partially processed due to an error, the unprocessed keys are returned as part of the invocation result inside the *unprocessedKeys* block.
- `BatchPutItem` is limited to 25 items.
- This operation **is not** supported when used with conflict detection. Using both at the same time may result in an error.

For the following example request mapping template:

```
{
  "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"
          }
        }
    ],
  }
}
```

The invocation result available in `$ctx.result` is as follows:

```
{
   "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": []
    }
}
```

The `$ctx.error` contains details about the error. The keys **data**, **unprocessedItems**, and each table key that was provided in the request mapping template are guaranteed to be present in the invocation result. Items that have been inserted are in the **data** block. Items that haven't been processed are marked as *null* inside the data block and are placed inside the **unprocessedItems** block.

For a more complete example, follow the DynamoDB Batch tutorial with AppSync here Tutorial: DynamoDB batch resolvers.

TransactGetItems

The `TransactGetItems` request mapping document lets you to tell the AWS AppSync DynamoDB resolver to make a `TransactGetItems` request to DynamoDB to retrieve multiple items, potentially across multiple tables. For this request template, you must specify the following:

- The table name of each request item where to retrieve the item from
- The key of each request item to retrieve from each table

The DynamoDB `TransactGetItems` limits apply and **no condition expression** can be provided.

The `TransactGetItems` mapping document has the following structure:

```
{
    "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" : {
                 ...
            }
        }
    ]
}
```

The fields are defined as follows:

TransactGetItems fields

TransactGetItems fields list

`version`

The template definition version. Only `2018-05-29` is supported. This value is required.

`operation`

The DynamoDB operation to perform. To perform the `TransactGetItems` DynamoDB operation, this must be set to `TransactGetItems`. This value is required.

`transactItems`

The request items to include. The value is an array of request items. At least one request item must be provided. This `transactItems` value is required.

`table`

The DynamoDB table to retrieve the item from. The value is a string of the table name. This `table` value is required.

`key`

The DynamoDB key representing the primary key of the item to retrieve. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping).

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

Things to remember:

- If a transaction succeeds, the order of retrieved items in the `items` block will be the same as the order of request items.
- Transactions are performed in an all-or-nothing way. If any request item causes an error, the whole transaction will not be performed and error details will be returned.
- A request item being unable to be retrieved is not an error. Instead, a *null* element appears in the *items* block in the corresponding position.
- If the error of a transaction is *TransactionCanceledException*, the `cancellationReasons` block will be populated. The order of cancellation reasons in `cancellationReasons` block will be the same as the order of request items.
- `TransactGetItems` is limited to 100 request items.

For the following example request mapping template:

```
{
    "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
            }
        }
    }
]
}
```

If the transaction succeeds and only the first requested item is retrieved, the invocation result available in `$ctx.result` is as follows:

```
{
    "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
}
```

If the transaction fails due to *TransactionCanceledException* caused by the first request item, the invocation result available in `$ctx.result` is as follows:

```
{
    "items": null,
    "cancellationReasons": [
       {
           "type":"Sample error type",
           "message":"Sample error message"
```

```
        },
        {
            "type":"None",
            "message":"None"
        }
    ]
}
```

The `$ctx.error` contains details about the error. The keys **items** and **cancellationReasons** are guaranteed to be present in `$ctx.result`.

For a more complete example, follow the DynamoDB Transaction tutorial with AppSync here Tutorial: DynamoDB transaction resolvers.

TransactWriteItems

The `TransactWriteItems` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `TransactWriteItems` request to DynamoDB to write multiple items, potentially to multiple tables. For this request template, you must specify the following:

- The destination table name of each request item
- The operation of each request item to perform. There are four types of operations that are supported: *PutItem*, *UpdateItem*, *DeleteItem*, and *ConditionCheck*
- The key of each request item to write

The DynamoDB `TransactWriteItems` limits apply.

The `TransactWriteItems` mapping document has the following structure:

```
{
    "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 fields

TransactWriteItems fields list

The fields are defined as follows:

`version`

The template definition version. Only `2018-05-29` is supported. This value is required.

`operation`

The DynamoDB operation to perform. To perform the `TransactWriteItems` DynamoDB operation, this must be set to `TransactWriteItems`. This value is required.

`transactItems`

The request items to include. The value is an array of request items. At least one request item must be provided. This `transactItems` value is required.

For `PutItem`, the fields are defined as follows:

`table`

The destination DynamoDB table. The value is a string of the table name. This `table` value is required.

`operation`

The DynamoDB operation to perform. To perform the `PutItem` DynamoDB operation, this must be set to `PutItem`. This value is required.

`key`

The DynamoDB key representing the primary key of the item to put. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping). This value is required.

`attributeValues`

The rest of the attributes of the item to be put into DynamoDB. For more information about how to specify a “typed value”, see Type system (request mapping). This field is optional.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `PutItem` request overwrites any existing entry for that item. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see Transaction condition expressions. This value is optional.

For `UpdateItem`, the fields are defined as follows:

`table`

The DynamoDB table to update. The value is a string of the table name. This `table` value is required.

`operation`

The DynamoDB operation to perform. To perform the `UpdateItem` DynamoDB operation, this must be set to `UpdateItem`. This value is required.

`key`

The DynamoDB key representing the primary key of the item to update. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a "typed value", see Type system (request mapping). This value is required.

`update`

The `update` section lets you specify an update expression that describes how to update the item in DynamoDB. For more information about how to write update expressions, see the DynamoDB UpdateExpressions documentation. This section is required.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `UpdateItem` request updates the existing entry regardless of its current state. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see Transaction condition expressions. This value is optional.

For `DeleteItem`, the fields are defined as follows:

`table`

The DynamoDB table in which to delete the item. The value is a string of the table name. This `table` value is required.

`operation`

The DynamoDB operation to perform. To perform the `DeleteItem` DynamoDB operation, this must be set to `DeleteItem`. This value is required.

key

The DynamoDB key representing the primary key of the item to delete. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping). This value is required.

condition

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `DeleteItem` request deletes an item regardless of its current state. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see Transaction condition expressions. This value is optional.

For `ConditionCheck`, the fields are defined as follows:

table

The DynamoDB table in which to check the condition. The value is a string of the table name. This `table` value is required.

operation

The DynamoDB operation to perform. To perform the `ConditionCheck` DynamoDB operation, this must be set to `ConditionCheck`. This value is required.

key

The DynamoDB key representing the primary key of the item to condition check. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping). This value is required.

condition

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see Transaction condition expressions. This value is required.

Things to remember:

- Only keys of request items are returned in the response, if successful. The order of keys will be the same as the order of request items.
- Transactions are performed in an all-or-nothing way. If any request item causes an error, the whole transaction will not be performed and error details will be returned.
- No two request items can target the same item. Otherwise they will cause *TransactionCanceledException* error.
- If the error of a transaction is *TransactionCanceledException*, the `cancellationReasons` block will be populated. If a request item's condition check fails **and** you did not specify `returnValuesOnConditionCheckFailure` to be `false`, the item existing in the table will be retrieved and stored in `item` at the corresponding position of `cancellationReasons` block.
- `TransactWriteItems` is limited to 100 request items.
- This operation **is not** supported when used with conflict detection. Using both at the same time may result in an error.

For the following example request mapping template:

```
{
    "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"
                    }
                }
            },
        }
    ]
}
```

If the transaction succeeds, the invocation result available in `$ctx.result` is as follows:

```
{
    "keys": [
       // Key of the PutItem request
       {
           "post_id": "p1",
       },
       // Key of the UpdateItem request
       {
           "author_id": "a1"
       }
    ],
    "cancellationReasons": null
}
```

If the transaction fails due to condition check failure of the `PutItem` request, the invocation result available in `$ctx.result` is as follows:

```
{
    "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"
       }
    ]
}
```

The `$ctx.error` contains details about the error. The keys **keys** and **cancellationReasons** are guaranteed to be present in `$ctx.result`.

For a more complete example, follow the DynamoDB Transaction tutorial with AppSync here Tutorial: DynamoDB transaction resolvers.

Type system (request mapping)

When using the AWS AppSync DynamoDB resolver to call your DynamoDB tables, AWS AppSync needs to know the type of each value to use in that call. This is because DynamoDB supports more type primitives than GraphQL or JSON (such as sets and binary data). AWS AppSync needs some hints when translating between GraphQL and DynamoDB, otherwise it would have to make some assumptions on how data is structured in your table.

For more information about DynamoDB data types, see the DynamoDB Data type descriptors and Data types documentation.

A DynamoDB value is represented by a JSON object containing a single key-value pair. The key specifies the DynamoDB type, and the value specifies the value itself. In the following example, the key S denotes that the value is a string, and the value `identifier` is the string value itself.

```
{ "S" : "identifier" }
```

Note that the JSON object cannot have more than one key-value pair. If more than one key-value pair is specified, the request mapping document isn't parsed.

A DynamoDB value is used anywhere in a request mapping document where you need to specify a value. Some places where you need to do this include: `key` and `attributeValue` sections, and the `expressionValues` section of expression sections. In the following example, the DynamoDB String value `identifier` is being assigned to the `id` field in a `key` section (perhaps in a `GetItem` request mapping document).

```
"key" : {
   "id" : { "S" : "identifier" }
}
```

Supported Types

AWS AppSync supports the following DynamoDB scalar, document, and set types:

String type S

A single string value. A DynamoDB String value is denoted by:

```
{ "S" : "some string" }
```

An example usage is:

```
"key" : {
   "id" : { "S" : "some string" }
}
```

String set type SS

A set of string values. A DynamoDB String Set value is denoted by:

```
{ "SS" : [ "first value", "second value", ... ] }
```

An example usage is:

```
"attributeValues" : {
   "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
}
```

Number type N

A single numeric value. A DynamoDB Number value is denoted by:

```
{ "N" : 1234 }
```

An example usage is:

```
"expressionValues" : {
   ":expectedVersion" : { "N" : 1 }
}
```

Number set type NS

A set of number values. A DynamoDB Number Set value is denoted by:

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

An example usage is:

```
"attributeValues" : {
   "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }
}
```

Binary type B

A binary value. A DynamoDB Binary value is denoted by:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

Note that the value is actually a string, where the string is the base64-encoded representation of the binary data. AWS AppSync decodes this string back into its binary value before sending it to DynamoDB. AWS AppSync uses the base64 decoding scheme as defined by RFC 2045: any character that isn't in the base64 alphabet is ignored.

An example usage is:

```
"attributeValues" : {
   "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

Binary set type BS

A set of binary values. A DynamoDB Binary Set value is denoted by:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

Note that the value is actually a string, where the string is the base64-encoded representation of the binary data. AWS AppSync decodes this string back into its binary value before sending it to DynamoDB. AWS AppSync uses the base64 decoding scheme as defined by RFC 2045: any character that is not in the base64 alphabet is ignored.

An example usage is:

```
"attributeValues" : {
   "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }
}
```

Boolean type BOOL

A Boolean value. A DynamoDB Boolean value is denoted by:

```
{ "BOOL" : true }
```

Note that only `true` and `false` are valid values.

An example usage is:

```
"attributeValues" : {
   "orderComplete" : { "BOOL" : false }
}
```

List type L

A list of any other supported DynamoDB value. A DynamoDB List value is denoted by:

```
{ "L" : [ ... ] }
```

Note that the value is a compound value, where the list can contain zero or more of any supported DynamoDB value (including other lists). The list can also contain a mix of different types.

An example usage is:

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

Map type `M`

Representing an unordered collection of key-value pairs of other supported DynamoDB values. A DynamoDB Map value is denoted by:

```
{ "M" : { ... } }
```

Note that a map can contain zero or more key-value pairs. The key must be a string, and the value can be any supported DynamoDB value (including other maps). The map can also contain a mix of different types.

An example usage is:

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

Null type `NULL`

A null value. A DynamoDB Null value is denoted by:

```
{ "NULL" : null }
```

An example usage is:

```
"attributeValues" : {
   "phoneNumbers" : { "NULL" : null }
}
```

For more information about each type, see the DynamoDB documentation .

Type system (response mapping)

When receiving a response from DynamoDB, AWS AppSync automatically converts it into GraphQL and JSON primitive types. Each attribute in DynamoDB is decoded and returned in the response mapping context.

For example, if DynamoDB returns the following:

```
{
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

Then the AWS AppSync DynamoDB resolver converts it into GraphQL and JSON types as:

```
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

This section explains how AWS AppSync converts the following DynamoDB scalar, document, and set types:

String type S

A single string value. A DynamoDB String value is returned as a string.

For example, if DynamoDB returned the following DynamoDB String value:

```
{ "S" : "some string" }
```

AWS AppSync converts it to a string:

```
"some string"
```

String set type SS

A set of string values. A DynamoDB String Set value is returned as a list of strings.

For example, if DynamoDB returned the following DynamoDB String Set value:

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync converts it to a list of strings:

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

Number type N

A single numeric value. A DynamoDB Number value is returned as a number.

For example, if DynamoDB returned the following DynamoDB Number value:

```
{ "N" : 1234 }
```

AWS AppSync converts it to a number:

```
1234
```

Number set type NS

A set of number values. A DynamoDB Number Set value is returned as a list of numbers.

For example, if DynamoDB returned the following DynamoDB Number Set value:

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync converts it to a list of numbers:

```
[ 67.8, 12.2, 70 ]
```

Binary type B

A binary value. A DynamoDB Binary value is returned as a string containing the base64 representation of that value.

For example, if DynamoDB returned the following DynamoDB Binary value:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync converts it to a string containing the base64 representation of the value:

```
"SGVsbG8sIFdvcmxkIQo="
```

Note that the binary data is encoded in the base64 encoding scheme as specified in RFC 4648 and RFC 2045.

Binary set type BS

A set of binary values. A DynamoDB Binary Set value is returned as a list of strings containing the base64 representation of the values.

For example, if DynamoDB returned the following DynamoDB Binary Set value:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync converts it to a list of strings containing the base64 representation of the values:

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

Note that the binary data is encoded in the base64 encoding scheme as specified in RFC 4648 and RFC 2045.

Boolean type BOOL

A Boolean value. A DynamoDB Boolean value is returned as a Boolean.

For example, if DynamoDB returned the following DynamoDB Boolean value:

```
{ "BOOL" : true }
```

AWS AppSync converts it to a Boolean:

```
true
```

List type L

A list of any other supported DynamoDB value. A DynamoDB List value is returned as a list of values, where each inner value is also converted.

For example, if DynamoDB returned the following DynamoDB List value:

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

AWS AppSync converts it to a list of converted values:

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

Map type `M`

A key/value collection of any other supported DynamoDB value. A DynamoDB Map value is returned as a JSON object, where each key/value is also converted.

For example, if DynamoDB returned the following DynamoDB Map value:

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
```

```
}
```

AWS AppSync converts it to a JSON object:

```
{
   "someString" : "A string value",
   "someNumber" : 1,
   "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

Null type `NULL`

A null value.

For example, if DynamoDB returned the following DynamoDB Null value:

```
{ "NULL" : null }
```

AWS AppSync converts it to a null:

```
null
```

Filters

When querying objects in DynamoDB using the `Query` and `Scan` operations, you can optionally specify a `filter` that evaluates the results and returns only the desired values.

The filter mapping section of a `Query` or `Scan` mapping document has the following structure:

```
"filter" : {
    "expression" : "filter expression"
    "expressionNames" : {
        "#name" : "name",
    },
    "expressionValues" : {
        ":value" : ... typed value
    },
}
```

The fields are defined as follows:

`expression`

The query expression. For more information about how to write filter expressions, see the DynamoDB QueryFilter and DynamoDB ScanFilter documentation. This field must be specified.

`expressionNames`

The substitutions for expression attribute *name* placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the `expression`. The value must be a string that corresponds to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the `expression`.

`expressionValues`

The substitutions for expression attribute *value* placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the `expression`, and the value must be a typed value. For more information about how to specify a "typed value", see Type System (Request Mapping). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the `expression`.

Example

The following example is a filter section for a mapping template, where entries retrieved from DynamoDB are only returned if the title starts with the `title` argument.

```
"filter" : {
    "expression" : "begins_with(#title, :title)",
    "expressionNames" : {
        "#title" : "title"
    },
    "expressionValues" : {
        ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
    }
}
```

Condition expressions

When you mutate objects in DynamoDB by using the `PutItem`, `UpdateItem`, and `DeleteItem` DynamoDB operations, you can optionally specify a condition expression that controls whether the

request should succeed or not, based on the state of the object already in DynamoDB before the operation is performed.

The AWS AppSync DynamoDB resolver allows a condition expression to be specified in `PutItem`, `UpdateItem`, and `DeleteItem` request mapping documents, and also a strategy to follow if the condition fails and the object was not updated.

Example 1

The following `PutItem` mapping document doesn't have a condition expression. As a result, it puts an item in DynamoDB even if an item with the same key already exists, thereby overwriting the existing item.

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   }
}
```

Example 2

The following `PutItem` mapping document does have a condition expression that allows the operation succeed only if an item with the same key does *not* exist in DynamoDB.

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   },
   "condition" : {
      "expression" : "attribute_not_exists(id)"
   }
}
```

By default, if the condition check fails, the AWS AppSync DynamoDB resolver returns an error for the mutation. However, the AWS AppSync DynamoDB resolver offers some additional features to help developers handle some common edge cases:

- If AWS AppSync DynamoDB resolver can determine that the current value in DynamoDB matches the desired result, it treats the operation as if it succeeded anyway.
- Instead of returning an error, you can configure the resolver to invoke a custom Lambda function to decide how the AWS AppSync DynamoDB resolver should handle the failure.

These are described in greater detail in the Handling a Condition Check Failure section.

For more information about DynamoDB conditions expressions, see the DynamoDB ConditionExpressions documentation .

Specifying a condition

The `PutItem`, `UpdateItem`, and `DeleteItem` request mapping documents all allow an optional `condition` section to be specified. If omitted, no condition check is made. If specified, the condition must be true for the operation to succeed.

A `condition` section has the following structure:

```
"condition" : {
    "expression" : "someExpression"
    "expressionNames" : {
        "#foo" : "foo"
    },
    "expressionValues" : {
        ":bar" : ... typed value
    },
    "equalsIgnore" : [ "version" ],
    "consistentRead" : true,
    "conditionalCheckFailedHandler" : {
        "strategy" : "Custom",
        "lambdaArn" : "arn:..."
    }
}
```

The following fields specify the condition:

`expression`

The update expression itself. For more information about how to write condition expressions, see the DynamoDB ConditionExpressions documentation . This field must be specified.

`expressionNames`

The substitutions for expression attribute name placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the *expression*, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the *expression*.

`expressionValues`

The substitutions for expression attribute value placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the expression, and the value must be a typed value. For more information about how to specify a "typed value", see Type System (Request Mapping). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the expression.

The remaining fields tell the AWS AppSync DynamoDB resolver how to handle a condition check failure:

`equalsIgnore`

When a condition check fails when using the `PutItem` operation, the AWS AppSync DynamoDB resolver compares the item currently in DynamoDB against the item it tried to write. If they are the same, it treats the operation as it if succeeded anyway. You can use the `equalsIgnore` field to specify a list of attributes that AWS AppSync should ignore when performing that comparison. For example, if the only difference was a `version` attribute, it treats the operation as if it succeeded. This field is optional.

`consistentRead`

When a condition check fails, AWS AppSync gets the current value of the item from DynamoDB using a strongly consistent read. You can use this field to tell the AWS AppSync DynamoDB resolver to use an eventually consistent read instead. This field is optional, and defaults to `true`.

`conditionalCheckFailedHandler`

This section allows you to specify how the AWS AppSync DynamoDB resolver treats a condition check failure after it has compared the current value in DynamoDB against the expected result. This section is optional. If omitted, it defaults to a strategy of `Reject`.

`strategy`

The strategy the AWS AppSync DynamoDB resolver takes after it has compared the current value in DynamoDB against the expected result. This field is required and has the following possible values:

`Reject`

The mutation fails, and an error is added to the GraphQL response.

`Custom`

The AWS AppSync DynamoDB resolver invokes a custom Lambda function to decide how to handle the condition check failure. When the `strategy` is set to `Custom`, the `lambdaArn` field must contain the ARN of the Lambda function to invoke.

`lambdaArn`

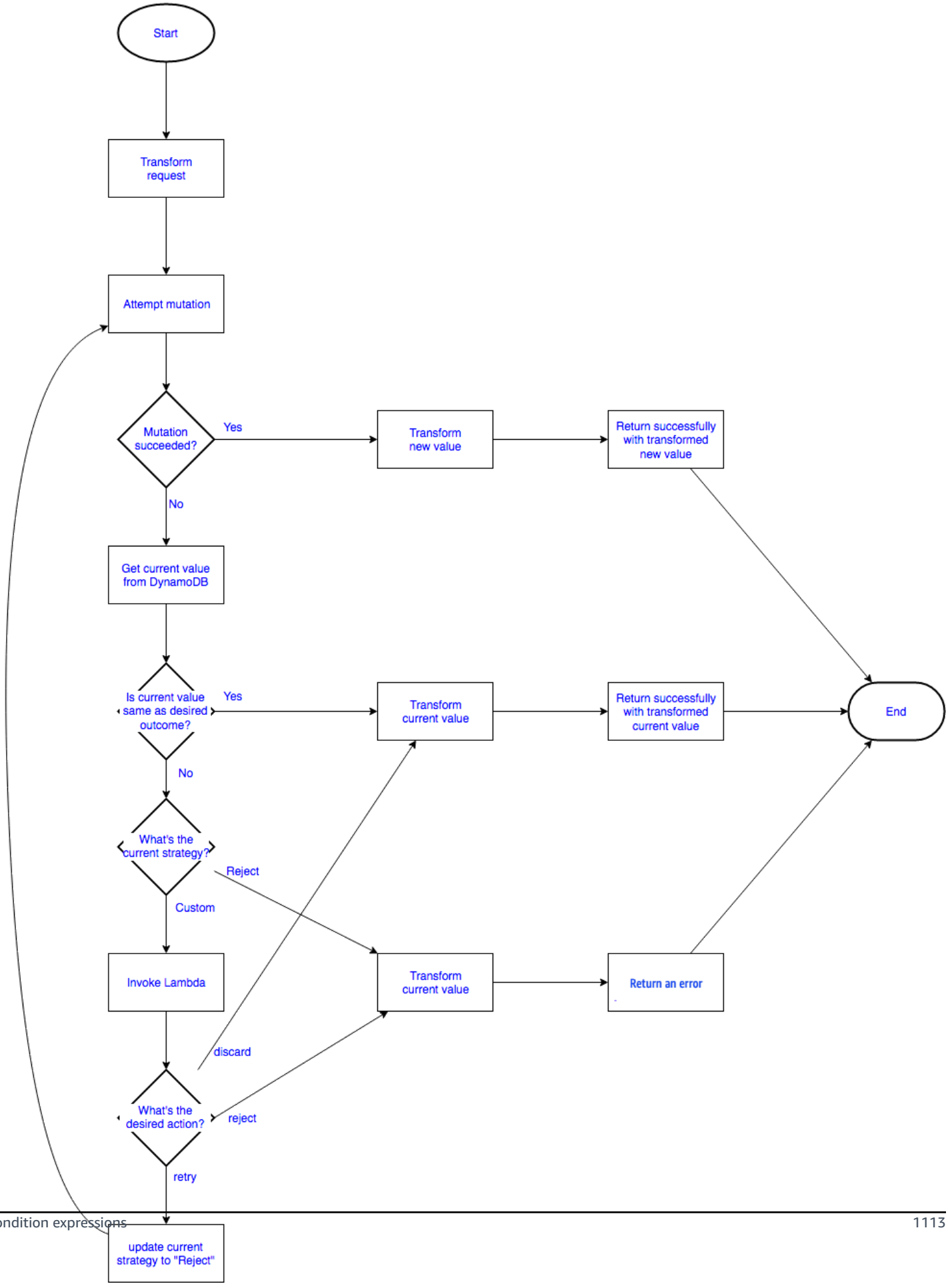
The ARN of the Lambda function to invoke that determines how the AWS AppSync DynamoDB resolver should handle the condition check failure. This field must only be specified when `strategy` is set to `Custom`. For more information about how to use this feature, see Handling a Condition Check Failure.

Handling a condition check failure

By default, when a condition check fails, the AWS AppSync DynamoDB resolver returns an error for the mutation and the current value of the object in DynamoDB. However, the AWS AppSync DynamoDB resolver offers some additional features to help developers handle some common edge cases:

- If AWS AppSync DynamoDB resolver can determine that the current value in DynamoDB matches the desired result, it treats the operation as if it succeeded anyway.
- Instead of returning an error, you can configure the resolver to invoke a custom Lambda function to decide how the AWS AppSync DynamoDB resolver should handle the failure.

The flowchart for this process is:


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"

Checking for the desired result

When the condition check fails, the AWS AppSync DynamoDB resolver performs a `GetItem` DynamoDB request to get the current value of the item from DynamoDB. By default, it uses a strongly consistent read, however this can be configured using the `consistentRead` field in the `condition` block and compare it against the expected result:

- For the `PutItem` operation, the AWS AppSync DynamoDB resolver compares the current value against the one it attempted to write, excluding any attributes listed in `equalsIgnore` from the comparison. If the items are the same, it treats the operation as successful and returns the item that was retrieved from DynamoDB. Otherwise, it follows the configured strategy.

  For example, if the `PutItem` request mapping document looked like the following:

```
{
   "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" ]
   }
}
```

  And the item currently in DynamoDB looked like the following:

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

The AWS AppSync DynamoDB resolver would compare the item it tried to write against the current value, see that the only difference was the `version` field, but because it's configured to ignore the `version` field, it treats the operation as successful and returns the item that was retrieved from DynamoDB.

- For the `DeleteItem` operation, the AWS AppSync DynamoDB resolver checks to verify that an item was returned from DynamoDB. If no item was returned, it treats the operation as successful. Otherwise, it follows the configured strategy.
- For the `UpdateItem` operation, the AWS AppSync DynamoDB resolver does not have enough information to determine if the item currently in DynamoDB matches the expected result, and therefore follows the configured strategy.

If the current state of the object in DynamoDB is different from the expected result, the AWS AppSync DynamoDB resolver follows the configured strategy, to either reject the mutation or invoke a Lambda function to determine what to do next.

Following the "reject" strategy

When following the `Reject` strategy, the AWS AppSync DynamoDB resolver returns an error for the mutation.

For example, given the following mutation request:

```
mutation {
    updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
        Name
        theVersion
    }
}
```

If the item returned from DynamoDB looks like the following:

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

And the response mapping template looks like the following:

```
{
   "id" : $util.toJson($context.result.id),
   "Name" : $util.toJson($context.result.name),
   "theVersion" : $util.toJson($context.result.version)
}
```

The GraphQL response looks like the following:

```
{
  "data": null,
  "errors": [
    {
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      ...
    }
  ]
}
```

Also, if any fields in the returned object are filled by other resolvers and the mutation had succeeded, they won't be resolved when the object is returned in the `error` section.

Following the "custom" strategy

When following the `Custom` strategy, the AWS AppSync DynamoDB resolver invokes a Lambda function to decide what to do next. The Lambda function chooses one of the following options:

- `reject` the mutation. This tells the AWS AppSync DynamoDB resolver to behave as if the configured strategy was `Reject`, returning an error for the mutation and the current value of the object in DynamoDB as described in the previous section.
- `discard` the mutation. This tells the AWS AppSync DynamoDB resolver to silently ignore the condition check failure and returns the value in DynamoDB.
- `retry` the mutation. This tells the AWS AppSync DynamoDB resolver to retry the mutation with a new request mapping document.

The Lambda invocation request

The AWS AppSync DynamoDB resolver invokes the Lambda function specified in the `lambdaArn`. It uses the same `service-role-arn` configured on the data source. The payload of the invocation has the following structure:

```
{
    "arguments": { ... },
    "requestMapping": {... },
    "currentValue": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

The fields are defined as follows:

`arguments`

The arguments from the GraphQL mutation. This is the same as the arguments available to the request mapping document in `$context.arguments`.

`requestMapping`

The request mapping document for this operation.

`currentValue`

The current value of the object in DynamoDB.

`resolver`

Information about the AWS AppSync resolver.

`identity`

Information about the caller. This is the same as the identity information available to the request mapping document in `$context.identity`.

A full example of the payload:

```
{
    "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"
    }
}
```

The Lambda Invocation Response

The Lambda function can inspect the invocation payload and apply any business logic to decide how the AWS AppSync DynamoDB resolver should handle the failure. There are three options for handling the condition check failure:

- `reject` the mutation. The response payload for this option must have this structure:

```
{
    "action": "reject"
}
```

  This tells the AWS AppSync DynamoDB resolver to behave as if the configured strategy was `Reject`, returning an error for the mutation and the current value of the object in DynamoDB, as described in the section above.

- `discard` the mutation. The response payload for this option must have this structure:

```
{
    "action": "discard"
}
```

  This tells the AWS AppSync DynamoDB resolver to silently ignore the condition check failure and returns the value in DynamoDB.

- `retry` the mutation. The response payload for this option must have this structure:

```
{
    "action": "retry",
    "retryMapping": { ... }
}
```

  This tells the AWS AppSync DynamoDB resolver to retry the mutation with a new request mapping document. The structure of the `retryMapping` section depends on the DynamoDB operation, and is a subset of the full request mapping document for that operation.

  For `PutItem`, the `retryMapping` section has the following structure. For a description of the `attributeValues` field, see PutItem.

```
{
    "attributeValues": { ... },
    "condition": {
        "equalsIgnore" = [ ... ],
        "consistentRead" = true
    }
}
```

For `UpdateItem`, the `retryMapping` section has the following structure. For a description of the `update` section, see UpdateItem.

```
{
    "update" : {
        "expression" : "someExpression"
        "expressionNames" : {
            "#foo" : "foo"
        },
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "condition": {
        "consistentRead" = true
    }
}
```

For `DeleteItem`, the `retryMapping` section has the following structure.

```
{
    "condition": {
        "consistentRead" = true
    }
}
```

There is no way to specify a different operation or key to work on. The AWS AppSync DynamoDB resolver only allows retries of the same operation on the same object. Also, the `condition` section doesn't allow a `conditionalCheckFailedHandler` to be specified. If the retry fails, the AWS AppSync DynamoDB resolver follows the `Reject` strategy.

Here is an example Lambda function to deal with a failed `PutItem` request. The business logic looks at who made the call. If it was made by `jeffTheAdmin`, it retries the request, updating the `version` and `expectedVersion` from the item currently in DynamoDB. Otherwise, it rejects the mutation.

```
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)
};
```

Transaction condition expressions

Transaction condition expressions are available in request mapping templates of all four types of operations in `TransactWriteItems`, namely, `PutItem`, `DeleteItem`, `UpdateItem`, and `ConditionCheck`.

For `PutItem`, `DeleteItem`, and `UpdateItem`, the transaction condition expression is optional. For `ConditionCheck`, the transaction condition expression is required.

Example 1

The following transactional `DeleteItem` mapping document does not have a condition expression. As a result, it deletes the item in DynamoDB.

```
{
```

```
   "version": "2018-05-29",
   "operation": "TransactWriteItems",
   "transactItems": [
      {
         "table": "posts",
         "operation": "DeleteItem",
         "key": {
            "id": { "S" : "1" }
         }
      }
   ]
}
```

Example 2

The following transactional `DeleteItem` mapping document does have a transaction condition expression that allows the operation succeed only if the author of that post equals a certain name.

```
{
   "version": "2018-05-29",
   "operation": "TransactWriteItems",
   "transactItems": [
      {
         "table": "posts",
         "operation": "DeleteItem",
         "key": {
            "id": { "S" : "1" }
         }
         "condition": {
            "expression": "author = :author",
            "expressionValues": {
               ":author": { "S" : "Chunyan" }
            }
         }
      }
   ]
}
```

If the condition check fails, it will cause `TransactionCanceledException` and the error detail will be returned in `$ctx.result.cancellationReasons`. Note that by default, the old item in DynamoDB that made condition check fail will be returned in `$ctx.result.cancellationReasons`.

Specifying a condition

The `PutItem`, `UpdateItem`, and `DeleteItem` request mapping documents all allow an optional `condition` section to be specified. If omitted, no condition check is made. If specified, the condition must be true for the operation to succeed. The `ConditionCheck` must have a `condition` section to be specified. The condition must be true for the whole transaction to succeed.

A `condition` section has the following structure:

```
"condition": {
    "expression": "someExpression",
    "expressionNames": {
        "#foo": "foo"
    },
    "expressionValues": {
        ":bar": ... typed value
    },
    "returnValuesOnConditionCheckFailure": false
}
```

The following fields specify the condition:

`expression`

The update expression itself. For more information about how to write condition expressions, see the DynamoDB ConditionExpressions documentation . This field must be specified.

`expressionNames`

The substitutions for expression attribute name placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the *expression*, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the *expression*.

`expressionValues`

The substitutions for expression attribute value placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the expression, and the value must be a typed value. For more information about how to specify a “typed value”, see Type System (request

mapping). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the expression.

`returnValuesOnConditionCheckFailure`

Specify whether to retrieve the item in DynamoDB back when a condition check fails. The retrieved item will be in `$ctx.result.cancellationReasons[$index].item`, where `$index` is the index of the request item that failed the condition check. This value defaults to true.

Projections

When reading objects in DynamoDB using the `GetItem`, `Scan`, `Query`, `BatchGetItem`, and `TransactGetItems` operations, you can optionally specify a projection that identifies the attributes that you want. The projection has the following structure, which is similar to filters:

```
"projection" : {
    "expression" : "projection expression"
    "expressionNames" : {
        "#name" : "name",
    }
}
```

The fields are defined as follows:

`expression`

The projection expression, which is a string. To retrieve a single attribute, specify its name. For multiple attributes, the names must be comma-separated values. For more information on writing projection expressions, see the DynamoDB projection expressions documentation. This field is required.

`expressionNames`

The substitutions for expression attribute *name* placeholders in the form of key-value pairs. The key corresponds to a name placeholder used in the `expression`. The value must be a string that corresponds to the attribute name of the item in DynamoDB. This field is optional and should only be populated with substitutions for expression attribute name placeholders used in the `expression`. For more information about `expressionNames`, see the DynamoDB documentation.

Example 1

The following example is a projection section for a VTL mapping template in which only the attributes `author` and `id` are returned from DynamoDB:

```
"projection" : {
    "expression" : "#author, id",
    "expressionNames" : {
        "#author" : "author"
    }
}
```

Tip

You can access your GraphQL request selection set using $context.info.selectionSetList. This field allows you to frame your projection expression dynamically according to your requirements.

Note

While using projection expressions with the `Query` and `Scan` operations, the value for `select` must be `SPECIFIC_ATTRIBUTES`. For more information, see the DynamoDB documentation.

AWS AppSync resolver mapping template reference for RDS

The AWS AppSync RDS resolver mapping templates allow developers to send SQL queries to a Data API for Amazon Aurora Serverless and get back the result of these queries.

Request mapping template

The RDS request mapping template is fairly simple:

```
{
    "version": "2018-05-29",
    "statements": [],
    "variableMap": {},
```

```
    "variableTypeHintMap": {}
}
```

Here is the JSON schema representation of the RDS request mapping template, once resolved.

```
{
    "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"
        }
    }
}
```

The following is an example of the request mapping template with a static query:

```
{
    "version": "2018-05-29",
    "statements": [
        "select title, isbn13 from BOOKS where author = 'Mark Twain'"
    ]
}
```

Version

Common to all request mapping templates, the version field defines the version that the template uses. The version field is required. The value “2018-05-29” is the only version supported for the Amazon RDS mapping templates.

```
"version": "2018-05-29"
```

Statements and VariableMap

The statements array is a placeholder for the developer-provided queries. Currently, up to two queries per request mapping template are supported. The `variableMap` is an optional field that contains aliases that can be used to make the SQL statements shorter and more readable. For example, the following is possible:

```
{
```

```
"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 will use the variable map values to construct the **SqlParameterized** queries that will be sent to the Amazon Aurora Serverless Data API. The SQL statements are executed with parameters provided in the variable map, which eliminates the risk of SQL injection.

VariableTypeHintMap

The `variableTypeHintMap` is an optional field containing aliased types that can be used to send SQL parameter type hints. These type hints avoid explicit casting in the SQL statements, making them shorter. For example, the following is possible:

```
{
    "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 will use the variable map value to construct the queries that are sent to the Amazon Aurora Serverless Data API. It also uses the `variableTypeHintMap` data and sends the type's information to RDS. RDS-supported `typeHints` can be found here.

AWS AppSync resolver mapping template reference for OpenSearch

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

The AWS AppSync resolver for Amazon OpenSearch Service enables you to use GraphQL to store and retrieve data in existing OpenSearch Service domains in your account. This resolver works by allowing you to map an incoming GraphQL request into an OpenSearch Service request, then map the OpenSearch Service response back to GraphQL. This section describes the mapping templates for the supported OpenSearch Service operations.

Request mapping template

Most OpenSearch Service request mapping templates have a common structure where just a few pieces change. The following example runs a search against an OpenSearch Service domain, where documents are organized under an index called `post`. The search parameters are defined in the `body` section, with many of the common query clauses being defined in the `query` field. This example will search for documents containing `"Nadia"`, or `"Bailey"`, or both, in the `author` field of a document:

```
{
    "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" }}
```

```
                    ]
                }
            }
        }
    }
}
```

Response mapping template

As with other data sources, OpenSearch Service sends a response to AWS AppSync that needs to be converted to GraphQL. .

Most GraphQL queries are looking for the `_source` field from an OpenSearch Service response. Because you can do searches to return either an individual document or a list of documents, there are two common response mapping templates used in OpenSearch Service:

List of Results

```
[
    #foreach($entry in $context.result.hits.hits)
      #if( $velocityCount > 1 ) , #end
        $utils.toJson($entry.get("_source"))
    #end
]
```

Individual Item

```
$utils.toJson($context.result.get("_source"))
```

`operation` field

> **Note**
>
> This applies only to the Request mapping template.

HTTP method or verb (GET, POST, PUT, HEAD or DELETE) that AWS AppSync sends to the OpenSearch Service domain. Both the key and the value must be a string.

```
"operation" : "PUT"
```

`path` field

> **Note**
>
> This applies only to the Request mapping template.

The search path for an OpenSearch Service request from AWS AppSync. This forms a URL for the operation's HTTP verb. Both the key and the value must be strings.

```
"path" : "/<indexname>/_doc/<_id>"
"path" : "/<indexname>/_doc"
"path" : "/<indexname>/_search"
"path" : "/<indexname>/_update/<_id>
```

When the mapping template is evaluated, this path is sent as part of the HTTP request, including the OpenSearch Service domain. For example, the previous example might translate to:

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

`params` field

> **Note**
>
> This applies only to the Request mapping template.

Used to specify what action your search performs, most commonly by setting the **query** value inside of the **body**. However, there are several other capabilities that can be configured, such as the formatting of responses.

- **headers**

  The header information, as key-value pairs. Both the key and the value must be strings. For example:

  ```
  "headers" : {
      "Content-Type" : "application/json"
  ```

```
}
```

> **Note**
>
> AWS AppSync currently supports only JSON as a `Content-Type`.

- **queryString**

  Key-value pairs that specify common options, such as code formatting for JSON responses. Both the key and the value must be a string. For example, if you want to get pretty-formatted JSON, you would use:

```
"queryString" : {
    "pretty" : "true"
}
```

- **body**

  This is the main part of your request, allowing AWS AppSync to craft a well-formed search request to your OpenSearch Service domain. The key must be a string comprised of an object. A couple of demonstrations are shown below.

Example 1

Return all documents with a city matching “seattle”:

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "match" : {
            "city" : "seattle"
        }
    }
}
```

Example 2

Return all documents matching “washington” as the city or the state:

```
"body":{
```

```
    "from":0,
    "size":50,
    "query" : {
        "multi_match" : {
            "query" : "washington",
            "fields" : ["city", "state"]
        }
    }
}
```

Passing variables

> **Note**
>
> This applies only to the Request mapping template.

You can also pass variables as part of evaluation in the VTL statement. For example, suppose you had a GraphQL query such as the following:

```
query {
    searchForState(state: "washington"){
        ...
    }
}
```

The mapping template could take the state as an argument:

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "multi_match" : {
            "query" : "$context.arguments.state",
            "fields" : ["city", "state"]
        }
    }
}
```

For a list of utilities you can include in the VTL, see Access Request Headers.

AWS AppSync resolver mapping template reference for Lambda

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

You can use AWS AppSync functions and resolvers to invoke Lambda functions located in your account. You can shape your request payloads and the response from your Lambda functions before returning them to your clients. You can also use mapping templates to give hints to AWS AppSync about the nature of the operation to be invoked. This section describes the different mapping templates for the supported Lambda operations.

Request mapping template

The Lambda request mapping template handles fields related to your Lambda function:

```
{
  "version": string,
  "operation": Invoke|BatchInvoke,
  "payload": any type,
  "invocationType": RequestResponse|Event
}
```

This is the JSON schema representation of the Lambda request mapping template when resolved:

```
{
  "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
}
```

Here's an example that uses an `invoke` operation with its payload data being the `getPost` field from a GraphQL schema along with its arguments from the context:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $util.toJson($context.arguments)
  }
}
```

The entire mapping document is passed as the input to your Lambda function so that the previous example now looks like this:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "id": "postId1"
    }
  }
}
```

Version

Common to all request mapping templates, the `version` defines the version that the template uses. The `version` is required and is a static value:

```
"version": "2018-05-29"
```

Operation

The Lambda data source lets you define two operations in the `operation` field: `Invoke` and `BatchInvoke`. The `Invoke` operation lets AWS AppSync know to call your Lambda function for every GraphQL field resolver. `BatchInvoke` instructs AWS AppSync to batch requests for the current GraphQL field. The `operation` field is required.

For `Invoke`, the resolved request mapping template matches the input payload of the Lambda function. Let's modify the example above:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
    "payload": {
      "arguments": $util.toJson($context.arguments)
    }
}
```

This is resolved and passed to the Lambda function, which could look something like this:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
    "payload": {
      "arguments": {
        "id": "postId1"
      }
    }
}
```

For `BatchInvoke`, the mapping template is applied to every field resolver in the batch. For conciseness, AWS AppSync merges all the resolved mapping template `payload` values into a list under a single object matching the mapping template. The following example template shows the merge:

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": $util.toJson($context)
}
```

This template is resolved into the following mapping document:

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

Each element of the `payload` list corresponds to a single batch item. The Lambda function is also expected to return a list-shaped response matching the order of the items sent in the request:

```
[
  { "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
]
```

Payload

The `payload` field is a container used to pass any well-formed JSON to the Lambda function. If the `operation` field is set to `BatchInvoke`, AWS AppSync wraps the existing `payload` values into a list. The `payload` field is optional.

Invocation type

The Lambda data source allows you to define two invocation types: `RequestResponse` and `Event`. The invocation types are synonymous with the invocation types defined in the Lambda API. The `RequestResponse` invocation type lets AWS AppSync call your Lambda function synchronously to wait for a response. The `Event` invocation allows you to invoke your Lambda function asynchronously. For more information on how Lambda handles `Event` invocation type requests, see Asynchronous invocation. The `invocationType` field is optional. If this field is not included in the request, AWS AppSync will default to the `RequestResponse` invocation type.

For any `invocationType` field, the resolved request matches the input payload of the Lambda function. Let's modify the example above:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event"
  "payload": {
    "arguments": $util.toJson($context.arguments)
  }
}
```

This is resolved and passed to the Lambda function, which could look something like this:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
```

```
    }
  }
}
```

When the `BatchInvoke` operation is used in conjunction with the `Event` invocation type field, AWS AppSync merges the field resolver in the same way mentioned above, and the request is passed to your Lambda function as an asynchronous event with the `payload` being a list of values. We recommend that you disable resolver caching for `Event` invocation type resolvers because these would not be sent to Lambda if there were a cache hit.

Response mapping template

As with other data sources, your Lambda function sends a response to AWS AppSync that must be converted to a GraphQL type.

The result of the Lambda function is set on the `context` object that is available via the Velocity Template Language (VTL) `$context.result` property.

If the shape of your Lambda function response exactly matches the shape of the GraphQL type, you can forward the response using the following response mapping template:

```
$util.toJson($context.result)
```

There are no required fields or shape restrictions that apply to the response mapping template. However, because GraphQL is strongly typed, the resolved mapping template must match the expected GraphQL type.

Lambda function batched response

If the `operation` field is set to `BatchInvoke`, AWS AppSync expects a list of items back from the Lambda function. In order for AWS AppSync to map each result back to the original request item, the response list must match in size and order. It's valid to have `null` items in the response list; `$ctx.result` is set to *null* accordingly.

Direct Lambda Resolvers

If you wish to circumvent the use of mapping templates entirely, AWS AppSync can provide a default payload to your Lambda function and a default Lambda function response to a GraphQL type. You can choose to provide a request template, a response template, or neither, and AWS AppSync handles it accordingly.

Direct Lambda request mapping template

When the request mapping template is not provided, AWS AppSync will send the `Context` object directly to your Lambda function as an `Invoke` operation. For more information about the structure of the `Context` object, see AWS AppSync resolver mapping template context reference.

Direct Lambda response mapping template

When the response mapping template is not provided, AWS AppSync does one of two things upon receiving your Lambda function's response. If you did not provide a request mapping template or if you provided a request mapping template with the version `2018-05-29`, the response will be equivalent to the following response mapping template:

```
#if($ctx.error)
     $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
 #end
 $util.toJson($ctx.result)
```

If you provided a template with the version `2017-02-28`, the response logic functions equivalently to the following response mapping template:

```
$util.toJson($ctx.result)
```

Superficially, the mapping template bypass operates similarly to using certain mapping templates as shown in the preceding examples. However, behind the scenes, the evaluation of the mapping templates is circumvented entirely. Because the template evaluation step is bypassed, applications might experience less overhead and latency during the response in some scenarios compared to a Lambda function with a response mapping template that needs to be evaluated.

Custom error handling in Direct Lambda Resolver responses

You can customize error responses from Lambda functions that Direct Lambda Resolvers invoke by raising a custom exception. The following example demonstrates how to create a custom exception using JavaScript:

```
class CustomException extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomException";
```

```
  }
}

throw new CustomException("Custom message");
```

When exceptions are raised, the `errorType` and `errorMessage` are the `name` and `message`, respectively, of the custom error that is thrown.

If `errorType` is `UnauthorizedException`, AWS AppSync returns the default message (`"You are not authorized to make this call."`) instead of a custom message.

The following snippet is an example GraphQL response that demonstrates a custom `errorType`:

```
{
  "data": {
    "query": null
  },
  "errors": [
    {
      "path": [
        "query"
      ],
      "data": null,
      "errorType": "CustomException",
      "errorInfo": null,
      "locations": [
        {
          "line": 5,
          "column": 10,
          "sourceName": null
        }
      ],
      "message": "Custom Message"
    }
  ]
}
```

Direct Lambda Resolvers: Batching enabled

You can enable batching for your Direct Lambda Resolver by configuring the `maxBatchSize` on your resolver. When `maxBatchSize` is set to a value greater than `0` for a Direct Lambda resolver, AWS AppSync sends requests in batches to your Lambda function in sizes up to `maxBatchSize`.

Setting `maxBatchSize` to `0` on a Direct Lambda resolver turns off batching.

For more information on how batching with Lambda resolvers works, see Advanced use case: Batching.

Request mapping template

When batching is enabled and the request mapping template is not provided, AWS AppSync sends a list of `Context` objects as a `BatchInvoke` operation directly to your Lambda function.

Response mapping template

When batching is enabled and the response mapping template is not provided, the response logic is equivalent to the following response mapping template:

```
#if( $context.result && $context.result.errorMessage )
      $utils.error($context.result.errorMessage, $context.result.errorType,
      $context.result.data)
#else
      $utils.toJson($context.result.data)
#end
```

The Lambda function must return a list of results in the same order as the list of `Context` objects that were sent. You can return individual errors by providing an `errorMessage` and `errorType` for a specific result. Each result in the list has the following format:

```
{
   "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**
>
> Other fields in the result object are currently ignored.

Handling errors from Lambda

You can return an error for all results by throwing an exception or an error in your Lambda function. If the payload request or response size for your batch request is too large, Lambda

returns an error. In that case, you should consider reducing your `maxBatchSize` or reducing the size of the response payload.

For information on handling individual errors, see Returning individual errors.

Sample Lambda functions

Using the schema below, you can create a Direct Lambda Resolver for the `Post.relatedPosts` field resolver and enable batching by setting `maxBatchSize` above `0`:

```
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]
}
```

In the following query, the Lambda function will be called with batches of requests to resolve `relatedPosts`:

```
query getAllPosts {
  allPosts {
    id
    relatedPosts {
```

```
      id
    }
  }
}
```

A simple implementation of a Lambda function is provided below:

```
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' }
    }
  })
}
```

AWS AppSync resolver mapping template reference for EventBridge

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

The AWS AppSync resolver mapping template used with the EventBridge data source allows you to send custom events to the Amazon EventBridge bus.

Request mapping template

The `PutEvents` request mapping template allows you to send multiple custom events to an EventBridge event bus. The mapping document has the following structure:

```
{
    "version" : "2018-05-29",
    "operation" : "PutEvents",
    "events" : [{}]
}
```

The following is an example of a request mapping template for EventBridge:

```
{
    "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"
    }

    ]
}
```

Response mapping template

If the `PutEvents` operation is successful, the response from EventBridge is included in the `$ctx.result`:

```
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end
  $util.toJson($ctx.result)
```

Errors that occur while performing `PutEvents` operations such as `InternalExceptions` or `Timeouts` will appear in `$ctx.error`. For a list of EventBridge's common errors, see the EventBridge common error reference.

The `result` will be in the following format:

```
{
    "Entries" [
        {
            "ErrorCode" : String,
            "ErrorMessage" : String,
            "EventId" : String
        }
    ],
    "FailedEntryCount" : number
}
```

- **Entries**

The ingested event results, both successful and unsuccessful. If the ingestion was successful, the entry has the `EventID` in it. Otherwise, you can use the `ErrorCode` and `ErrorMessage` to identify the problem with the entry.

For each record, the index of the response element is the same as the index in the request array.

- **FailedEntryCount**

  The number of failed entries. This value is represented as an integer.

For more information about the response of `PutEvents`, see PutEvents.

Example sample response 1

The following example is a `PutEvents` operation with two successful events:

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ],
    "FailedEntryCount" : 0
}
```

Example sample response 2

The following example is a `PutEvents` operation with three events, two successes and one fail:

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        },
        {
            "ErrorCode" : "SampleErrorCode",
```

```
            "ErrorMessage" : "Sample Error Message"
        }
    ],
    "FailedEntryCount" : 1
}
```

PutEvents fields

`PutEvents` contains the following mapping template fields:

- **Version**

  Common to all request mapping templates, the `version` field defines the version that the template uses. This field is required. The value `2018-05-29` is the only version supported for the EventBridge mapping templates.

- **Operation**

  The only supported operation is `PutEvents`. This operation allows you to add custom events to your event bus.

- **Events**

  An array of events that will be added to the event bus. This array should have an allocation of 1 - 10 items.

  The `Event` object is a valid JSON object that has the following fields:

  - `"source"`: A string that defines the source of the event.
  - `"detail"`: A JSON object that you can use to attach information about the event. This field can be an empty map (`{ }`).
  - `"detailType`: A string that identifies the type of event.
  - `"resources"`: A JSON array of strings that identifies resources involved in the event. This field can be an empty array.
  - `"time"`: The event timestamp provided as a string. This should follow the RFC3339 timestamp format.

The snippets below are some examples of valid `Event` objects:

Example 1

```
{
    "source" : "source1",
    "detail" : {
        "key1" : [1,2,3,4],
        "key2" : "strval"
    },
    "detailType" : "sampleDetailType",
    "resources" : ["Resouce1", "Resource2"],
    "time" : "2022-01-10T05:00:10Z"
}
```

Example 2

```
{
    "source" : "source1",
    "detail" : {},
    "detailType" : "sampleDetailType"
}
```

Example 3

```
{
    "source" : "source1",
    "detail" : {
        "key1" : 1200
    },
    "detailType" : "sampleDetailType",
    "resources" : []
}
```

AWS AppSync resolver mapping template reference for None data source

Note

We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

The AWS AppSync resolver mapping template used with the data source of type *None*, enables you to shape requests for AWS AppSync local operations.

Request mapping template

The mapping template is simple and enables you to pass as much context information as possible via the `payload` field.

```
{
   "version": string,
   "payload": any type
}
```

Here is the JSON schema representation of the request mapping template, once resolved:

```
{
    "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
}
```

Here is an example where the field arguments are passed via the VTL context property `$context.arguments`:

```
{
```

```
    "version": "2018-05-29",
    "payload": $util.toJson($context.arguments)
}
```

The value of the `payload` field will be forwarded to the response mapping template and available on the VTL context property (`$context.result`).

This is an example representing the interpolated value of the `payload` field:

```
{
    "id": "postId1"
}
```

Version

Common to all request mapping templates, the `version` field defines the version used by the template.

The `version` field is required.

Example:

```
"version": "2018-05-29"
```

Payload

The `payload` field is a container that can be used to pass any well-formed JSON to the response mapping template.

The `payload` field is optional.

Response mapping template

Because there is no data source, the value of the `payload` field will be forwarded to the response mapping template and set on the `context` object that is available via the VTL `$context.result` property.

If the shape of the `payload` field value exactly matches the shape of the GraphQL type, you can forward the response using the following response mapping template:

```
$util.toJson($context.result)
```

There are no required fields or shape restrictions that apply to the response mapping template. However, because GraphQL is strongly typed, the resolved mapping template must match the expected GraphQL type.

AWS AppSync resolver mapping template reference for HTTP

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

The AWS AppSync HTTP resolver mapping templates enable you to send requests from AWS AppSync to any HTTP endpoint, and responses from your HTTP endpoint back to AWS AppSync. By using mapping templates, you can provide hints to AWS AppSync about the nature of the operation to be invoked. This section describes the different mapping templates for the supported HTTP resolver.

Request mapping template

```
{
    "version": "2018-05-29",
    "method": "PUT|POST|GET|DELETE|PATCH",
    "params": {
        "query": Map,
        "headers": Map,
        "body": any
    },
    "resourcePath": string
}
```

After the HTTP request mapping template is resolved, the JSON schema representation of the request mapping template looks like the following:

```
{
    "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
    "type": "object",
    "properties": {
        "version": {
        "$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"
    ]
}
```

Following is an example of an HTTP POST request, with a `text/plain` body:

```
{
    "version": "2018-05-29",
    "method": "POST",
    "params": {
        "headers":{
        "Content-Type":"text/plain"
        },
        "body":"this is an example of text body"
    },
    "resourcePath": "/"
}
```

Version

> **Note**
>
> This applies only to the Request mapping template.

Defines the version that the template uses. `version` is common to all request mapping templates and is required.

```
"version": "2018-05-29"
```

Method

> **Note**
>
> This applies only to the Request mapping template.

HTTP method or verb (GET, POST, PUT, PATCH, or DELETE) that AWS AppSync sends to the HTTP endpoint.

```
"method": "PUT"
```

ResourcePath

> **Note**
>
> This applies only to the Request mapping template.

The resource path that you want to access. Along with the endpoint in the HTTP data source, the resource path forms the URL that the AWS AppSync service makes a request to.

```
"resourcePath": "/v1/users"
```

When the mapping template is evaluated, this path is sent as part of the HTTP request, including the HTTP endpoint. For example, the previous example might translate to the following:

```
PUT <endpoint>/v1/users
```

Params fields

> **Note**
>
> This applies only to the Request mapping template.

Used to specify what action your search performs, most commonly by setting the **query** value inside the **body**. However, there are several other capabilities that can be configured, such as the formatting of responses.

headers

The header information, as key-value pairs. Both the key and the value must be strings.

For example:

```
"headers" : {
    "Content-Type" : "application/json"
}
```

Currently supported `Content-Type` headers are:

```
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
```

Note: You can't set the following HTTP headers:

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

Key-value pairs that specify common options, such as code formatting for JSON responses. Both the key and the value must be a string. The following example shows how you can send a query string as `?type=json`:

```
"query" : {
    "type" : "json"
}
```

body

The body contains the HTTP request body that you choose to set. The request body is always a UTF-8 encoded string unless the content type specifies the charset.

```
"body":"body string"
```

Response

See an example here.

Certificate Authorities (CA) Recognized by AWS AppSync for HTTPS Endpoints

> **Note**
>
> Let's Encrypt is accepted via the *identrust* and *isrgrootx1* certificates. No action on your part is required if you use Let's Encrypt.

At this time, self-signed certificates are not supported by HTTP resolvers when using HTTPS. AWS AppSync recognizes the following Certificate Authorities when resolving SSL/TLS certificates for HTTPS:

Known root certificates in AWS AppSync

Name	Date	SHA1 Fingerprint
digicertassuredidrootca	*Apr 21, 2018*	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
trustcenterclass2caii	*Apr 21, 2018*	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
thawtepremiumserverca	*Apr 21, 2018*	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	*Nov 23, 2016*	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
swisssignplatinumg2ca	*Apr 21, 2018*	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66
swisssignsilverg2ca	*Apr 21, 2018*	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
thawteserverca	*Apr 21, 2018*	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
equifaxsecureebusinessca1	*Apr 21, 2018*	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
securetrustca	*Apr 21, 2018*	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
utnuserfirstclientauthemailca	*Apr 21, 2018*	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
thawtepersonalfreemailca	*Apr 21, 2018*	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
affirmtrustnetworkingca	*Apr 21, 2018*	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F

Name	Date	SHA1 Fingerprint
entrustevca	*Apr 21, 2018*	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
utnuserfirsthardwareca	*Apr 21, 2018*	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
certumca	*Apr 21, 2018*	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
addtrustclass1ca	*Apr 21, 2018*	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
entrustrootcag2	*Apr 21, 2018*	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
equifaxsecureca	*Apr 21, 2018*	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
quovadisrootca3	*Apr 21, 2018*	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	*Apr 21, 2018*	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
digicertglobalrootg2	*Apr 21, 2018*	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
digicerthighassurancceevrootca	*Apr 21, 2018*	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
secomvalicertclass1ca	*Apr 21, 2018*	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
equifaxsecureglobalebusinessca1	*Apr 21, 2018*	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36

Name	Date	SHA1 Fingerprint
geotrustuniversalca	*Apr 21, 2018*	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
deprecateditsecca	*Jan 27, 2012*	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
verisignclass3ca	*Apr 21, 2018*	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
thawteprimaryrootcag3	*Apr 21, 2018*	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
thawteprimaryrootcag2	*Apr 21, 2018*	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
deutschetelekomrootca2	*Apr 21, 2018*	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
buypassclass3ca	*Apr 21, 2018*	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
utnuserfirstobjectca	*Apr 21, 2018*	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46
geotrustprimaryca	*Apr 21, 2018*	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
buypassclass2ca	*Apr 21, 2018*	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
baltimorecodesigningca	*Apr 21, 2018*	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D
verisignclass1ca	*Apr 21, 2018*	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1

Name	Date	SHA1 Fingerprint
baltimorecybertrustca	*Apr 21, 2018*	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
starfieldclass2ca	*Apr 21, 2018*	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
camerfirmachamberscommerceca	*Apr 21, 2018*	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
ttelesecglobalrootclass3ca	*Apr 21, 2018*	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
verisignclass3g5ca	*Apr 21, 2018*	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
ttelesecglobalrootclass2ca	*Apr 21, 2018*	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
trustcenteruniversalcai	*Apr 21, 2018*	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
verisignclass3g4ca	*Apr 21, 2018*	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3g3ca	*Apr 21, 2018*	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
xrampglobalca	*Apr 21, 2018*	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
amzninternalrootca	*Dec 12, 2008*	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06
certplusclass3pprimaryca	*Apr 21, 2018*	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79

Name	Date	SHA1 Fingerprint
certumtrustednetworkca	*Apr 21, 2018*	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
verisignclass3g2ca	*Apr 21, 2018*	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
globalsignr3ca	*Apr 21, 2018*	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
utndatacorpsgcca	*Apr 21, 2018*	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
secomscrootca2	*Apr 21, 2018*	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
gtecybertrustgloballca	*Apr 21, 2018*	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
secomscrootca1	*Apr 21, 2018*	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
affirmtrustcommercialca	*Apr 21, 2018*	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
trustcenterclass4caii	*Apr 21, 2018*	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
verisignuniversalrootca	*Apr 21, 2018*	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
globalsignr2ca	*Apr 21, 2018*	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
certplusclass2primaryca	*Apr 21, 2018*	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB

Name	Date	SHA1 Fingerprint
digicertglobalrootca	*Apr 21, 2018*	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
globalsignca	*Apr 21, 2018*	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
thawteprimaryrootca	*Apr 21, 2018*	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
starfieldrootg2ca	*Apr 21, 2018*	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
geotrustglobalca	*Apr 21, 2018*	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
soneraclass2ca	*Apr 21, 2018*	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
verisigntsaca	*Apr 21, 2018*	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
soneraclass1ca	*Apr 21, 2018*	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
quovadisrootca	*Apr 21, 2018*	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
affirmtrustpremiumeccca	*Apr 21, 2018*	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
starfieldservicesrootg2ca	*Apr 21, 2018*	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
valicertclass2ca	*Apr 21, 2018*	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6

Name	Date	SHA1 Fingerprint
comodoaaaca	*Apr 21, 2018*	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
aolrootca2	*Apr 21, 2018*	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
keynectisrootca	*Apr 21, 2018*	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
addtrustqualifiedca	*Apr 21, 2018*	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
aolrootca1	*Apr 21, 2018*	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignclass2g3ca	*Apr 21, 2018*	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
addtrustexternalca	*Apr 21, 2018*	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
verisignclass2g2ca	*Apr 21, 2018*	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
geotrustprimarycag3	*Apr 21, 2018*	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycag2	*Apr 21, 2018*	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
swisssigngoldg2ca	*Apr 21, 2018*	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
entrust2048ca	*Apr 21, 2018*	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31

Name	Date	SHA1 Fingerprint
chunghwaepkirootca	*Apr 21, 2018*	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
camerfirmachambersignca	*Apr 21, 2018*	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
camerfirmachambersca	*Apr 21, 2018*	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
godaddyclass2ca	*Apr 21, 2018*	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
affirmtrustpremiumca	*Apr 21, 2018*	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
verisignclass1g3ca	*Apr 21, 2018*	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
secomevrootca1	*Apr 21, 2018*	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
verisignclass1g2ca	*Apr 21, 2018*	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
amzninternalinfosecag3	*Feb 27, 2015*	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	*Nov 23, 2016*	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
godaddyrootg2ca	*Apr 21, 2018*	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
digicertassuredidrootca	*Apr 21, 2018*	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43

Name	Date	SHA1 Fingerprint
microseceszignorootca2009	*Apr 21, 2018*	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
affirmtrustcommercial	*Apr 21, 2018*	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
comodoecccertificationauthority	*Apr 21, 2018*	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
cadisigrootr2	*Apr 21, 2018*	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
swisssignsilvercag2	*Apr 21, 2018*	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
securetrustca	*Apr 21, 2018*	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
cadisigrootr1	*Apr 21, 2018*	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
accvraiz1	*Apr 21, 2018*	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
entrustrootcertificationauthority	*Apr 21, 2018*	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
camerfirmaglobalchambersignroot	*Apr 21, 2018*	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
dstacescax6	*Apr 21, 2018*	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
identrustpublicsectorrootca1	*Apr 21, 2018*	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD

Name	Date	SHA1 Fingerprint
starfieldrootcertificateauthorityg2	*Apr 21, 2018*	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
secureglobalca	*Apr 21, 2018*	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
eecertificationcentrerootca	*Apr 21, 2018*	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
opentrustrootcag3	*Apr 21, 2018*	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8A:A6
teliasonerarootcav1	*Apr 21, 2018*	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
autoridaddecertificacionfirmaprofesionalcifa62634068	*Apr 21, 2018*	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
opentrustrootcag2	*Apr 21, 2018*	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	*Apr 21, 2018*	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:D5:7E
globalsigneccrootcar5	*Apr 21, 2018*	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
globalsigneccrootcar4	*Apr 21, 2018*	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
izenpecom	*Apr 21, 2018*	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19

Name	Date	SHA1 Fingerprint
turktrustelektroniksertifikahizmetsaglayicisih5	*Apr 21, 2018*	C4:18:F6:4D:46:D1:DF:00:3D:27:30:13:72:43:A9:12:11:C6:75:FB
gdcatrustauthr5root	*Apr 21, 2018*	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
dtrustrootclass3ca22009	*Apr 21, 2018*	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
quovadisrootca3	*Apr 21, 2018*	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	*Apr 21, 2018*	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
geotrustprimarycertificationauthorityg3	*Apr 21, 2018*	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycertificationauthorityg2	*Apr 21, 2018*	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
oistewisekeyglobalrootgbca	*Apr 21, 2018*	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
addtrustexternalroot	*Apr 21, 2018*	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
chambersofcommerceroot2008	*Apr 21, 2018*	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
digicertglobalrootg3	*Apr 21, 2018*	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E

Name	Date	SHA1 Fingerprint
comodoaaaservicesroot	*Apr 21, 2018*	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
digicertglobalrootg2	*Apr 21, 2018*	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
certinomisrootca	*Apr 21, 2018*	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:88:A8
oistewisekeyglobalrootgaca	*Apr 21, 2018*	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
dstrootcax3	*Apr 21, 2018*	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
certigna	*Apr 21, 2018*	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
digicerthighassuranceevrootca	*Apr 21, 2018*	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
soneraclass2rootca	*Apr 21, 2018*	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
trustcorrootcertca2	*Apr 21, 2018*	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
usertrustrsacertificationauthority	*Apr 21, 2018*	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
trustcorrootcertca1	*Apr 21, 2018*	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
geotrustuniversalca	*Apr 21, 2018*	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79

Name	Date	SHA1 Fingerprint
certsignrootca	*Apr 21, 2018*	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
amazonrootca4	*Apr 21, 2018*	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
amazonrootca3	*Apr 21, 2018*	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
amazonrootca2	*Apr 21, 2018*	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
verisignuniversalrootcertificationauthority	*Apr 21, 2018*	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
amazonrootca1	*Apr 21, 2018*	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
networksolutionscertificateauthority	*Apr 21, 2018*	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
thawteprimaryrootcag3	*Apr 21, 2018*	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
affirmtrustnetworking	*Apr 21, 2018*	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
thawteprimaryrootcag2	*Apr 21, 2018*	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
trustcoreca1	*Apr 21, 2018*	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
deutschetelekomrootca2	*Apr 21, 2018*	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF

Name	Date	SHA1 Fingerprint
godaddyrootcertificateauthorityg2	*Apr 21, 2018*	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
entrustrootcertificationauthorityec1	*Apr 21, 2018*	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
szafirrootca2	*Apr 21, 2018*	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE
tubitakkamusmsslkoksertifikasisurum1	*Apr 21, 2018*	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
buypassclass3rootca	*Apr 21, 2018*	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
comodorsacertificationauthority	*Apr 21, 2018*	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
netlockaranyclassgoldfotanusitvany	*Apr 21, 2018*	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
securitycommunicationrootca2	*Apr 21, 2018*	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
dtrustrootclass3ca2ev2009	*Apr 21, 2018*	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
starfieldclass2ca	*Apr 21, 2018*	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
pscprocert	*Apr 21, 2018*	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
actalisauthenticationrootca	*Apr 21, 2018*	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC

Name	Date	SHA1 Fingerprint
staatdernederlandenrootcag3	*Apr 21, 2018*	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
cfcaevroot	*Apr 21, 2018*	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
digicerttrustedrootg4	*Apr 21, 2018*	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
staatdernederlandenrootcag2	*Apr 21, 2018*	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
securitycommunicationevrootca1	*Apr 21, 2018*	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
globalsignrootcar3	*Apr 21, 2018*	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
globalsignrootcar2	*Apr 21, 2018*	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
certumtrustednetworkca2	*Apr 21, 2018*	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
acraizfnmtrcm	*Apr 21, 2018*	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
hellenicacademicandresearchinstitutionseccrootca2015	*Apr 21, 2018*	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
certplusrootcag2	*Apr 21, 2018*	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:CC:1A
twcarootcertificationauthority	*Apr 21, 2018*	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48

Name	Date	SHA1 Fingerprint
twcaglobalrootca	*Apr 21, 2018*	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
certplusrootcag1	*Apr 21, 2018*	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:F7:66
geotrustuniversalca2	*Apr 21, 2018*	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
baltimorecybertrustroot	*Apr 21, 2018*	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
buypassclass2rootca	*Apr 21, 2018*	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
certumtrustednetworkca	*Apr 21, 2018*	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
digicertassuredidrootg3	*Apr 21, 2018*	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
digicertassuredidrootg2	*Apr 21, 2018*	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F
isrgrootx1	*Apr 21, 2018*	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
entrustnetpremium2048secureserverca	*Apr 21, 2018*	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
certplusclass2primaryca	*Apr 21, 2018*	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
digicertglobalrootca	*Apr 21, 2018*	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36

Name	Date	SHA1 Fingerprint
entrustrootcertificationauthorityg2	*Apr 21, 2018*	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
starfieldservicesrootcertificateauthorityg2	*Apr 21, 2018*	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
thawteprimaryrootca	*Apr 21, 2018*	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
atostrustedroot2011	*Apr 21, 2018*	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
geotrustglobalca	*Apr 21, 2018*	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
luxtrustglobalroot2	*Apr 21, 2018*	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5F:3F
etugracertificationauthority	*Apr 21, 2018*	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
visaecommerceroot	*Apr 21, 2018*	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
quovadisrootca	*Apr 21, 2018*	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
identrustcommercialrootca1	*Apr 21, 2018*	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
staatdernederlandenevrootca	*Apr 21, 2018*	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
ttelesecglobalrootclass3	*Apr 21, 2018*	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1

Name	Date	SHA1 Fingerprint
ttelesecglobalroot class2	*Apr 21, 2018*	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62: 32:17:65:CF:17:D8:94:E9
comodocertificatio nauthority	*Apr 21, 2018*	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C: BA:6A:BE:D1:F7:BD:EF:7B
securitycommunicat ionrootca	*Apr 21, 2018*	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38: 0F:C6:56:8F:5D:AC:B2:F7
quovadisrootca3g3	*Apr 21, 2018*	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D: 27:96:E6:A4:CF:22:2E:7D
xrampglobalcaroot	*Apr 21, 2018*	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30: 54:F3:4C:52:B7:E5:58:C6
securesignrootca11	*Apr 21, 2018*	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8: 5B:B1:C3:65:C7:D8:11:B3
affirmtrustpremium	*Apr 21, 2018*	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F: 7D:6A:06:65:26:32:28:27
globalsignrootca	*Apr 21, 2018*	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81: F2:15:01:52:A4:1D:82:9C
swisssigngoldcag2	*Apr 21, 2018*	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6: 45:25:3A:6F:9F:1A:27:61
quovadisrootca2g3	*Apr 21, 2018*	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38: 02:05:00:E1:25:F5:C8:36
affirmtrustpremium ecc	*Apr 21, 2018*	B8:23:6B:00:2F:1D:16:86:53:01:55:6C: 11:A4:37:CA:EB:FF:C3:BB
geotrustprimarycer tificatio nauthority	*Apr 21, 2018*	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2: 10:0D:D6:02:90:37:F0:96

Name	Date	SHA1 Fingerprint
quovadisrootca1g3	*Apr 21, 2018*	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67
hongkongpostrootca1	*Apr 21, 2018*	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
usertrustecccertificationauthority	*Apr 21, 2018*	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0
cybertrustglobalroot	*Apr 21, 2018*	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
godaddyclass2ca	*Apr 21, 2018*	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
hellenicacademicandresearchinstitutionsrootca2015	*Apr 21, 2018*	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
ecacc	*Apr 21, 2018*	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:F2:E8
hellenicacademicandresearchinstitutionsrootca2011	*Apr 21, 2018*	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
verisignclass3publicprimarycertificationauthorityg5	*Apr 21, 2018*	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5

Name	Date	SHA1 Fingerprint
verisignclass3publicprimarycertificationauthorityg4	*Apr 21, 2018*	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3publicprimarycertificationauthorityg3	*Apr 21, 2018*	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
trustisfpsrootca	*Apr 21, 2018*	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
epkirootcertificationauthority	*Apr 21, 2018*	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
globalchambersignroot2008	*Apr 21, 2018*	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
camerfirmachambersofcommerceroot	*Apr 21, 2018*	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert81.pem	*Mar 13, 2014*	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
mozillacert99.pem	*Mar 13, 2014*	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
mozillacert145.pem	*Mar 13, 2014*	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:3A:04
mozillacert37.pem	*Mar 13, 2014*	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert4.pem	*Mar 13, 2014*	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B

Name	Date	SHA1 Fingerprint
mozillacert70.pem	*Mar 13, 2014*	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
mozillacert88.pem	*Mar 13, 2014*	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
mozillacert134.pem	*Mar 13, 2014*	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
mozillacert26.pem	*Mar 13, 2014*	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
mozillacert77.pem	*Mar 13, 2014*	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
mozillacert123.pem	*Mar 13, 2014*	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10:DD:6B:DF:99:72:2C:96:E5
mozillacert15.pem	*Mar 13, 2014*	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
mozillacert66.pem	*Mar 13, 2014*	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34
mozillacert112.pem	*Mar 13, 2014*	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
mozillacert55.pem	*Mar 13, 2014*	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
mozillacert101.pem	*Mar 13, 2014*	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
mozillacert119.pem	*Mar 13, 2014*	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE

Name	Date	SHA1 Fingerprint
mozillacert44.pem	*Mar 13, 2014*	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
mozillacert108.pem	*Mar 13, 2014*	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
mozillacert95.pem	*Mar 13, 2014*	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert141.pem	*Mar 13, 2014*	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
mozillacert33.pem	*Mar 13, 2014*	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
mozillacert0.pem	*Mar 13, 2014*	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
mozillacert84.pem	*Mar 13, 2014*	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2
mozillacert130.pem	*Mar 13, 2014*	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert148.pem	*Mar 13, 2014*	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
mozillacert22.pem	*Mar 13, 2014*	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
mozillacert7.pem	*Mar 13, 2014*	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
mozillacert73.pem	*Mar 13, 2014*	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E

Name	Date	SHA1 Fingerprint
mozillacert137.pem	*Mar 13, 2014*	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A1:D1
mozillacert11.pem	*Mar 13, 2014*	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
mozillacert29.pem	*Mar 13, 2014*	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
mozillacert62.pem	*Mar 13, 2014*	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert126.pem	*Mar 13, 2014*	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
mozillacert18.pem	*Mar 13, 2014*	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
mozillacert51.pem	*Mar 13, 2014*	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
mozillacert69.pem	*Mar 13, 2014*	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
mozillacert115.pem	*Mar 13, 2014*	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert40.pem	*Mar 13, 2014*	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
mozillacert58.pem	*Mar 13, 2014*	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
mozillacert104.pem	*Mar 13, 2014*	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E

Name	Date	SHA1 Fingerprint
mozillacert91.pem	*Mar 13, 2014*	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
mozillacert47.pem	*Mar 13, 2014*	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
mozillacert80.pem	*Mar 13, 2014*	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert98.pem	*Mar 13, 2014*	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
mozillacert144.pem	*Mar 13, 2014*	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
mozillacert36.pem	*Mar 13, 2014*	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D
mozillacert3.pem	*Mar 13, 2014*	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
mozillacert87.pem	*Mar 13, 2014*	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert133.pem	*Mar 13, 2014*	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
mozillacert25.pem	*Mar 13, 2014*	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
mozillacert76.pem	*Mar 13, 2014*	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert122.pem	*Mar 13, 2014*	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68

Name	Date	SHA1 Fingerprint
mozillacert14.pem	*Mar 13, 2014*	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
mozillacert65.pem	*Mar 13, 2014*	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:35:FB
mozillacert111.pem	*Mar 13, 2014*	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
mozillacert129.pem	*Mar 13, 2014*	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
mozillacert54.pem	*Mar 13, 2014*	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
mozillacert100.pem	*Mar 13, 2014*	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
mozillacert118.pem	*Mar 13, 2014*	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45
mozillacert151.pem	*Mar 13, 2014*	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
mozillacert43.pem	*Mar 13, 2014*	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
mozillacert107.pem	*Mar 13, 2014*	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
mozillacert94.pem	*Mar 13, 2014*	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
mozillacert140.pem	*Mar 13, 2014*	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7

Name	Date	SHA1 Fingerprint
mozillacert32.pem	*Mar 13, 2014*	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C
mozillacert83.pem	*Mar 13, 2014*	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:EA:46
mozillacert147.pem	*Mar 13, 2014*	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
mozillacert21.pem	*Mar 13, 2014*	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
mozillacert39.pem	*Mar 13, 2014*	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
mozillacert6.pem	*Mar 13, 2014*	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
mozillacert72.pem	*Mar 13, 2014*	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
mozillacert136.pem	*Mar 13, 2014*	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert10.pem	*Mar 13, 2014*	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert28.pem	*Mar 13, 2014*	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
mozillacert61.pem	*Mar 13, 2014*	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
mozillacert79.pem	*Mar 13, 2014*	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27

Name	Date	SHA1 Fingerprint
mozillacert125.pem	*Mar 13, 2014*	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert17.pem	*Mar 13, 2014*	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert50.pem	*Mar 13, 2014*	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
mozillacert68.pem	*Mar 13, 2014*	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
mozillacert114.pem	*Mar 13, 2014*	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert57.pem	*Mar 13, 2014*	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
mozillacert103.pem	*Mar 13, 2014*	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert90.pem	*Mar 13, 2014*	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
mozillacert46.pem	*Mar 13, 2014*	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
mozillacert97.pem	*Mar 13, 2014*	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
mozillacert143.pem	*Mar 13, 2014*	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
mozillacert35.pem	*Mar 13, 2014*	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41

Name	Date	SHA1 Fingerprint
mozillacert2.pem	*Mar 13, 2014*	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
mozillacert86.pem	*Mar 13, 2014*	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
mozillacert132.pem	*Mar 13, 2014*	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert24.pem	*Mar 13, 2014*	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
mozillacert9.pem	*Mar 13, 2014*	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
mozillacert75.pem	*Mar 13, 2014*	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
mozillacert121.pem	*Mar 13, 2014*	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
mozillacert139.pem	*Mar 13, 2014*	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert13.pem	*Mar 13, 2014*	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
mozillacert64.pem	*Mar 13, 2014*	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
mozillacert110.pem	*Mar 13, 2014*	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert128.pem	*Mar 13, 2014*	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D

Name	Date	SHA1 Fingerprint
mozillacert53.pem	*Mar 13, 2014*	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
mozillacert117.pem	*Mar 13, 2014*	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
mozillacert150.pem	*Mar 13, 2014*	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
mozillacert42.pem	*Mar 13, 2014*	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
mozillacert106.pem	*Mar 13, 2014*	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
mozillacert93.pem	*Mar 13, 2014*	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17
mozillacert31.pem	*Mar 13, 2014*	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
mozillacert49.pem	*Mar 13, 2014*	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71
mozillacert82.pem	*Mar 13, 2014*	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
mozillacert146.pem	*Mar 13, 2014*	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A
mozillacert20.pem	*Mar 13, 2014*	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert38.pem	*Mar 13, 2014*	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36

Name	Date	SHA1 Fingerprint
mozillacert5.pem	*Mar 13, 2014*	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
mozillacert71.pem	*Mar 13, 2014*	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert89.pem	*Mar 13, 2014*	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
mozillacert135.pem	*Mar 13, 2014*	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert27.pem	*Mar 13, 2014*	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
mozillacert60.pem	*Mar 13, 2014*	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert78.pem	*Mar 13, 2014*	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert124.pem	*Mar 13, 2014*	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert16.pem	*Mar 13, 2014*	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
mozillacert67.pem	*Mar 13, 2014*	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
mozillacert113.pem	*Mar 13, 2014*	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
mozillacert56.pem	*Mar 13, 2014*	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2

Name	Date	SHA1 Fingerprint
mozillacert102.pem	*Mar 13, 2014*	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
mozillacert45.pem	*Mar 13, 2014*	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert109.pem	*Mar 13, 2014*	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
mozillacert96.pem	*Mar 13, 2014*	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
mozillacert142.pem	*Mar 13, 2014*	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
mozillacert34.pem	*Mar 13, 2014*	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
mozillacert1.pem	*Mar 13, 2014*	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
mozillacert85.pem	*Mar 13, 2014*	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
mozillacert131.pem	*Mar 13, 2014*	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
mozillacert149.pem	*Mar 13, 2014*	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert23.pem	*Mar 13, 2014*	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
mozillacert8.pem	*Mar 13, 2014*	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:6A:0F

Name	Date	SHA1 Fingerprint
mozillacert74.pem	*Mar 13, 2014*	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert120.pem	*Mar 13, 2014*	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
mozillacert138.pem	*Mar 13, 2014*	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
mozillacert12.pem	*Mar 13, 2014*	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
mozillacert63.pem	*Mar 13, 2014*	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
mozillacert127.pem	*Mar 13, 2014*	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert19.pem	*Mar 13, 2014*	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
mozillacert52.pem	*Mar 13, 2014*	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F
mozillacert116.pem	*Mar 13, 2014*	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert41.pem	*Mar 13, 2014*	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
mozillacert59.pem	*Mar 13, 2014*	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
mozillacert105.pem	*Mar 13, 2014*	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC

Name	Date	SHA1 Fingerprint
mozillacert92.pem	*Mar 13, 2014*	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0
mozillacert30.pem	*Mar 13, 2014*	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
mozillacert48.pem	*Mar 13, 2014*	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC
verisignc4g2.pem	*Mar 20, 2014*	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52
verisignc2g3.pem	*Mar 20, 2014*	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
verisignc1g6.pem	*Dec 31, 2014*	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
verisignc2g2.pem	*Mar 20, 2014*	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
verisignroot.pem	*Mar 20, 2014*	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
verisignc2g1.pem	*Mar 20, 2014*	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:14:2A
verisignc3g5.pem	*Mar 20, 2014*	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
verisignc1g3.pem	*Mar 20, 2014*	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
verisignc3g4.pem	*Mar 20, 2014*	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A

Name	Date	SHA1 Fingerprint
verisignc1g2.pem	*Mar 20, 2014*	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
verisignc3g3.pem	*Mar 20, 2014*	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
verisignc1g1.pem	*Mar 20, 2014*	90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:55:6F
verisignc3g2.pem	*Mar 20, 2014*	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
verisignc3g1.pem	*Mar 20, 2014*	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
verisignc2g6.pem	*Dec 31, 2014*	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F
verisignc4g3.pem	*Mar 20, 2014*	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
gdroot-g2.pem	*Dec 31, 2014*	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
gd-class2-root.pem	*Dec 31, 2014*	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
gd_bundle-g2.pem	*Dec 31, 2014*	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8
dstacescax6	*Jun 18, 2018*	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
gd_bundle-g2.pem	*Jun 18, 2018*	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8

Name	Date	SHA1 Fingerprint
verisignc4g3.pem	*Jun 18, 2018*	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
swisssignplatinumg2ca	*Apr 21, 2018*	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66
geotrustprimarycertificationauthorityg3	*Jun 18, 2018*	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycertificationauthorityg2	*Jun 18, 2018*	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
buypassclass2rootca	*Jun 18, 2018*	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
camerfirmachambersofcommerceroot	*Jun 18, 2018*	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert20.pem	*Jun 18, 2018*	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert12.pem	*Jun 18, 2018*	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
mozillacert90.pem	*Jun 18, 2018*	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
mozillacert82.pem	*Jun 18, 2018*	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
mozillacert140.pem	*Jun 18, 2018*	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert74.pem	*Jun 18, 2018*	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F

Name	Date	SHA1 Fingerprint
mozillacert132.pem	*Jun 18, 2018*	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert66.pem	*Jun 18, 2018*	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34
mozillacert124.pem	*Jun 18, 2018*	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert58.pem	*Jun 18, 2018*	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
securitycommunicationrootca2	*Jun 18, 2018*	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert116.pem	*Jun 18, 2018*	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert108.pem	*Jun 18, 2018*	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
certigna	*Jun 18, 2018*	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert3.pem	*Jun 18, 2018*	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
verisignc1g1.pem	*Jun 18, 2018*	90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:55:6F
verisignc4g2.pem	*Jun 18, 2018*	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52
deutschetelekomroottca2	*Jun 18, 2018*	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF

Name	Date	SHA1 Fingerprint
starfieldrootg2ca	*Apr 21, 2018*	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
comodoecccertificationauthority	*Jun 18, 2018*	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
digicertglobalrootg3	*Jun 18, 2018*	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E
digicertglobalrootg2	*Jun 18, 2018*	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
mozillacert11.pem	*Jun 18, 2018*	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
mozillacert81.pem	*Jun 18, 2018*	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
mozillacert73.pem	*Jun 18, 2018*	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
szafirrootca2	*Jun 18, 2018*	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE
mozillacert131.pem	*Jun 18, 2018*	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
ecacc	*Jun 18, 2018*	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:F2:E8
mozillacert65.pem	*Jun 18, 2018*	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:35:FB
turktrustelektroniksertifikahizmetsaglayicisih5	*Jun 18, 2018*	C4:18:F6:4D:46:D1:DF:00:3D:27:30:13:72:43:A9:12:11:C6:75:FB

Name	Date	SHA1 Fingerprint
mozillacert123.pem	*Jun 18, 2018*	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10:DD:6B:DF:99:72:2C:96:E5
mozillacert57.pem	*Jun 18, 2018*	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
mozillacert115.pem	*Jun 18, 2018*	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert49.pem	*Jun 18, 2018*	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71
mozillacert107.pem	*Jun 18, 2018*	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
verisignclass3g4ca	*Apr 21, 2018*	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
securetrustca	*Jun 18, 2018*	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
mozillacert2.pem	*Jun 18, 2018*	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
buypassclass2ca	*Apr 21, 2018*	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
secomscrootca2	*Apr 21, 2018*	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
secomscrootca1	*Apr 21, 2018*	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
trustisfpsrootca	*Jun 18, 2018*	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04

Name	Date	SHA1 Fingerprint
hongkongpostrootca1	*Jun 18, 2018*	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
certsignrootca	*Jun 18, 2018*	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
geotrustprimaryca	*Apr 21, 2018*	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
twcaglobalrootca	*Jun 18, 2018*	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
camerfirmachambersca	*Apr 21, 2018*	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
mozillacert10.pem	*Jun 18, 2018*	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert80.pem	*Jun 18, 2018*	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert72.pem	*Jun 18, 2018*	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
comodoaaaca	*Apr 21, 2018*	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert130.pem	*Jun 18, 2018*	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert64.pem	*Jun 18, 2018*	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
mozillacert122.pem	*Jun 18, 2018*	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68

Name	Date	SHA1 Fingerprint
mozillacert56.pem	*Jun 18, 2018*	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
equifaxsecureebusinessca1	*Apr 21, 2018*	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
camerfirmachambersignca	*Apr 21, 2018*	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert114.pem	*Jun 18, 2018*	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert48.pem	*Jun 18, 2018*	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC
pscprocert	*Jun 18, 2018*	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert106.pem	*Jun 18, 2018*	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
mozillacert1.pem	*Jun 18, 2018*	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
eecertificationcentrerootca	*Jun 18, 2018*	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
digicertglobalrootca	*Jun 18, 2018*	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
thawteprimaryrootcag3	*Jun 18, 2018*	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
thawteprimaryrootcag2	*Jun 18, 2018*	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12

Name	Date	SHA1 Fingerprint
entrustrootcertificationauthorityec1	*Jun 18, 2018*	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
valicertclass2ca	*Apr 21, 2018*	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
globalchambersignroot2008	*Jun 18, 2018*	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
amazonrootca4	*Jun 18, 2018*	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
gd-class2-root.pem	*Jun 18, 2018*	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
amazonrootca3	*Jun 18, 2018*	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
amazonrootca2	*Jun 18, 2018*	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
securitycommunicationrootca	*Jun 18, 2018*	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
amazonrootca1	*Jun 18, 2018*	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
acraizfnmtrcm	*Jun 18, 2018*	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
quovadisrootca3g3	*Jun 18, 2018*	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D
certplusrootcag2	*Jun 18, 2018*	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:CC:1A

Name	Date	SHA1 Fingerprint
certplusrootcag1	*Jun 18, 2018*	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:F7:66
mozillacert71.pem	*Jun 18, 2018*	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert63.pem	*Jun 18, 2018*	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
mozillacert121.pem	*Jun 18, 2018*	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
ttelesecglobalrootclass3ca	*Apr 21, 2018*	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
mozillacert55.pem	*Jun 18, 2018*	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
mozillacert113.pem	*Jun 18, 2018*	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
baltimorecybertrustca	*Apr 21, 2018*	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
mozillacert47.pem	*Jun 18, 2018*	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
mozillacert105.pem	*Jun 18, 2018*	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC
mozillacert39.pem	*Jun 18, 2018*	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
usertrustecccertificationauthority	*Jun 18, 2018*	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0

Name	Date	SHA1 Fingerprint
mozillacert0.pem	*Jun 18, 2018*	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
securitycommunicationevrootca1	*Jun 18, 2018*	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
verisignc3g5.pem	*Jun 18, 2018*	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
globalsignr3ca	*Apr 21, 2018*	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
trustcoreca1	*Jun 18, 2018*	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
equifaxsecureglobalebusinessca1	*Apr 21, 2018*	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
geotrustuniversalca	*Jun 18, 2018*	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
affirmtrustpremiumca	*Apr 21, 2018*	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
staatdernederlandenrootcag3	*Jun 18, 2018*	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
staatdernederlandenrootcag2	*Jun 18, 2018*	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
mozillacert70.pem	*Jun 18, 2018*	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
secomevrootca1	*Apr 21, 2018*	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D

Name	Date	SHA1 Fingerprint
geotrustglobalca	*Jun 18, 2018*	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert62.pem	*Jun 18, 2018*	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert120.pem	*Jun 18, 2018*	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
mozillacert54.pem	*Jun 18, 2018*	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
mozillacert112.pem	*Jun 18, 2018*	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
mozillacert46.pem	*Jun 18, 2018*	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
swisssigngoldcag2	*Jun 18, 2018*	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert104.pem	*Jun 18, 2018*	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E
mozillacert38.pem	*Jun 18, 2018*	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36
certplusclass3primaryca	*Apr 21, 2018*	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
entrustrootcertificationauthorityg2	*Jun 18, 2018*	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
godaddyrootg2ca	*Apr 21, 2018*	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B

Name	Date	SHA1 Fingerprint
cfcaevroot	*Jun 18, 2018*	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
verisignc3g4.pem	*Jun 18, 2018*	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
geotrustuniversalca2	*Jun 18, 2018*	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
starfieldservicesrootg2ca	*Apr 21, 2018*	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
digicerthighassuranceevrootca	*Jun 18, 2018*	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
entrustnetpremium2048secureserverca	*Jun 18, 2018*	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
camerfirmaglobalchambersignroot	*Jun 18, 2018*	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
verisignclass3g3ca	*Apr 21, 2018*	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
godaddyclass2ca	*Jun 18, 2018*	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
mozillacert61.pem	*Jun 18, 2018*	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
mozillacert53.pem	*Jun 18, 2018*	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
atostrustedroot2011	*Jun 18, 2018*	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21

Name	Date	SHA1 Fingerprint
mozillacert111.pem	*Jun 18, 2018*	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
staatdernederlandenevrootca	*Jun 18, 2018*	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
mozillacert45.pem	*Jun 18, 2018*	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert103.pem	*Jun 18, 2018*	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert37.pem	*Jun 18, 2018*	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert29.pem	*Jun 18, 2018*	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
izenpecom	*Jun 18, 2018*	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
comodorsacertificationauthority	*Jun 18, 2018*	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
mozillacert99.pem	*Jun 18, 2018*	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
mozillacert149.pem	*Jun 18, 2018*	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
utnuserfirstobjectca	*Apr 21, 2018*	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46
verisignc3g3.pem	*Jun 18, 2018*	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6

Name	Date	SHA1 Fingerprint
dstrootcax3	*Jun 18, 2018*	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
addtrustexternalroot	*Jun 18, 2018*	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
certumtrustednetworkca	*Jun 18, 2018*	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
affirmtrustpremiumecc	*Jun 18, 2018*	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
starfieldclass2ca	*Jun 18, 2018*	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
actalisauthenticationrootca	*Jun 18, 2018*	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
verisignclass2g3ca	*Apr 21, 2018*	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
isrgrootx1	*Jun 18, 2018*	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
godaddyrootcertificateauthorityg2	*Jun 18, 2018*	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
mozillacert60.pem	*Jun 18, 2018*	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
chunghwaepkirootca	*Apr 21, 2018*	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert52.pem	*Jun 18, 2018*	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F

Name	Date	SHA1 Fingerprint
microseceszignorootca2009	*Jun 18, 2018*	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
securesignrootca11	*Jun 18, 2018*	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert110.pem	*Jun 18, 2018*	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert44.pem	*Jun 18, 2018*	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
mozillacert102.pem	*Jun 18, 2018*	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
mozillacert36.pem	*Jun 18, 2018*	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D
mozillacert28.pem	*Jun 18, 2018*	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
baltimorecybertrustroot	*Jun 18, 2018*	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
amzninternalrootca	*Dec 12, 2008*	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06
mozillacert98.pem	*Jun 18, 2018*	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
mozillacert148.pem	*Jun 18, 2018*	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
verisignc3g2.pem	*Jun 18, 2018*	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F

Name	Date	SHA1 Fingerprint
quovadisrootca2g3	*Jun 18, 2018*	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
geotrustprimarycertificationauthority	*Jun 18, 2018*	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
opentrustrootcag3	*Jun 18, 2018*	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8A:A6
opentrustrootcag2	*Jun 18, 2018*	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	*Jun 18, 2018*	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:D5:7E
verisignclass3ca	*Apr 21, 2018*	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
globalsignca	*Apr 21, 2018*	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
ttelesecglobalrootclass2ca	*Apr 21, 2018*	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
verisignclass1g3ca	*Apr 21, 2018*	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
verisignuniversalrootca	*Apr 21, 2018*	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
soneraclass2ca	*Apr 21, 2018*	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
starfieldservicesrootcertificateauthorityg2	*Jun 18, 2018*	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F

Name	Date	SHA1 Fingerprint
mozillacert51.pem	*Jun 18, 2018*	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
mozillacert43.pem	*Jun 18, 2018*	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
mozillacert101.pem	*Jun 18, 2018*	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
mozillacert35.pem	*Jun 18, 2018*	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
globalsignr2ca	*Apr 21, 2018*	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
mozillacert27.pem	*Jun 18, 2018*	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
affirmtrustpremium	*Jun 18, 2018*	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert19.pem	*Jun 18, 2018*	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
mozillacert97.pem	*Jun 18, 2018*	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
netlockaranyclassgoldfotanusitvany	*Jun 18, 2018*	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
mozillacert89.pem	*Jun 18, 2018*	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
verisignroot.pem	*Jun 18, 2018*	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54

Name	Date	SHA1 Fingerprint
mozillacert147.pem	*Jun 18, 2018*	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
aolrootca2	*Apr 21, 2018*	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	*Nov 23, 2016*	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
aolrootca1	*Apr 21, 2018*	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignc3g1.pem	*Jun 18, 2018*	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert139.pem	*Jun 18, 2018*	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
soneraclass2rootca	*Jun 18, 2018*	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
swisssignsilverg2ca	*Apr 21, 2018*	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
thawteprimaryrootca	*Jun 18, 2018*	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
gdcatrustauthr5root	*Jun 18, 2018*	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
trustcenterclass4caii	*Apr 21, 2018*	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
usertrustrsacertificationauthority	*Jun 18, 2018*	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E

Name	Date	SHA1 Fingerprint
digicertassuredidrootg3	*Jun 18, 2018*	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
digicertassuredidrootg2	*Jun 18, 2018*	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F
mozillacert50.pem	*Jun 18, 2018*	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
mozillacert42.pem	*Jun 18, 2018*	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
mozillacert100.pem	*Jun 18, 2018*	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
mozillacert34.pem	*Jun 18, 2018*	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
affirmtrustcommercialca	*Apr 21, 2018*	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert26.pem	*Jun 18, 2018*	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
globalsigneccrootcar5	*Jun 18, 2018*	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
globalsigneccrootcar4	*Jun 18, 2018*	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
buypassclass3rootca	*Jun 18, 2018*	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert18.pem	*Jun 18, 2018*	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9

Name	Date	SHA1 Fingerprint
mozillacert96.pem	*Jun 18, 2018*	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
verisignc2g6.pem	*Jun 18, 2018*	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F
secomvalicertclass1ca	*Apr 21, 2018*	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert88.pem	*Jun 18, 2018*	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
accvraiz1	*Jun 18, 2018*	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert146.pem	*Jun 18, 2018*	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A
mozillacert138.pem	*Jun 18, 2018*	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
verisignclass3g2ca	*Apr 21, 2018*	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
dtrustrootclass3ca2ev2009	*Jun 18, 2018*	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
xrampglobalca	*Apr 21, 2018*	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
mozillacert9.pem	*Jun 18, 2018*	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
verisignuniversalrootcertificationauthority	*Jun 18, 2018*	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54

Name	Date	SHA1 Fingerprint
tubitakkamusmsslkoksertifikasisurum1	*Jun 18, 2018*	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
mozillacert41.pem	*Jun 18, 2018*	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
mozillacert33.pem	*Jun 18, 2018*	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
mozillacert25.pem	*Jun 18, 2018*	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
mozillacert17.pem	*Jun 18, 2018*	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert95.pem	*Jun 18, 2018*	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
affirmtrustpremiumeccca	*Apr 21, 2018*	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert87.pem	*Jun 18, 2018*	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert145.pem	*Jun 18, 2018*	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:3A:04
mozillacert79.pem	*Jun 18, 2018*	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert137.pem	*Jun 18, 2018*	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A1:D1
digicertassuredidrootca	*Jun 18, 2018*	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43

Name	Date	SHA1 Fingerprint
addtrustqualifiedca	*Apr 21, 2018*	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert129.pem	*Jun 18, 2018*	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
verisignclass2g2ca	*Apr 21, 2018*	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
baltimorecodesigningca	*Apr 21, 2018*	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D
luxtrustglobalroot2	*Jun 18, 2018*	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5F:3F
visaecommerceroot	*Jun 18, 2018*	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
oistewisekeyglobalrootgbca	*Jun 18, 2018*	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
mozillacert8.pem	*Jun 18, 2018*	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:6A:0F
comodocertificationauthority	*Jun 18, 2018*	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
cia-crt-g3-02-ca	*Nov 23, 2016*	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
verisignc1g6.pem	*Jun 18, 2018*	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
trustcenterclass2caii	*Apr 21, 2018*	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E

Name	Date	SHA1 Fingerprint
quovadisrootca1g3	*Jun 18, 2018*	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67
mozillacert40.pem	*Jun 18, 2018*	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
cadisigrootr2	*Jun 18, 2018*	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
cadisigrootr1	*Jun 18, 2018*	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
mozillacert32.pem	*Jun 18, 2018*	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C
utndatacorpsgcca	*Apr 21, 2018*	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
mozillacert24.pem	*Jun 18, 2018*	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
addtrustclass1ca	*Apr 21, 2018*	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
mozillacert16.pem	*Jun 18, 2018*	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
affirmtrustnetworkingca	*Apr 21, 2018*	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert94.pem	*Jun 18, 2018*	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
mozillacert86.pem	*Jun 18, 2018*	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2

Name	Date	SHA1 Fingerprint
mozillacert144.pem	*Jun 18, 2018*	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
mozillacert78.pem	*Jun 18, 2018*	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert136.pem	*Jun 18, 2018*	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert128.pem	*Jun 18, 2018*	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D
verisignclass1g2ca	*Apr 21, 2018*	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
hellenicacademicandresearchinstitutionsrootca2015	*Jun 18, 2018*	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
soneraclass1ca	*Apr 21, 2018*	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
hellenicacademicandresearchinstitutionsrootca2011	*Jun 18, 2018*	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
certumtrustednetworkca2	*Jun 18, 2018*	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
equifaxsecureca	*Apr 21, 2018*	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
thawteserverca	*Apr 21, 2018*	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79

Name	Date	SHA1 Fingerprint
mozillacert7.pem	*Jun 18, 2018*	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
affirmtrustnetworking	*Jun 18, 2018*	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
deprecateditsecca	*Jan 27, 2012*	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
globalsignrootcar3	*Jun 18, 2018*	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
globalsignrootcar2	*Jun 18, 2018*	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
quovadisrootca	*Jun 18, 2018*	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert31.pem	*Jun 18, 2018*	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
entrustrootcertificationauthority	*Jun 18, 2018*	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert23.pem	*Jun 18, 2018*	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
mozillacert15.pem	*Jun 18, 2018*	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
verisignc2g3.pem	*Jun 18, 2018*	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
mozillacert93.pem	*Jun 18, 2018*	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17

Name	Date	SHA1 Fingerprint
mozillacert151.pem	*Jun 18, 2018*	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
mozillacert85.pem	*Jun 18, 2018*	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
certplusclass2primaryca	*Jun 18, 2018*	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
mozillacert143.pem	*Jun 18, 2018*	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
mozillacert77.pem	*Jun 18, 2018*	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
mozillacert135.pem	*Jun 18, 2018*	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert69.pem	*Jun 18, 2018*	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
mozillacert127.pem	*Jun 18, 2018*	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert119.pem	*Jun 18, 2018*	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
geotrustprimarycag3	*Apr 21, 2018*	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
identrustpublicsectorrootca1	*Jun 18, 2018*	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
geotrustprimarycag2	*Apr 21, 2018*	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0

Name	Date	SHA1 Fingerprint
trustcorrootcertca2	*Jun 18, 2018*	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
mozillacert6.pem	*Jun 18, 2018*	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
trustcorrootcertca1	*Jun 18, 2018*	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
networksolutionscertificate authority	*Jun 18, 2018*	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
twcarootcertificationauthority	*Jun 18, 2018*	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
addtrustexternalca	*Apr 21, 2018*	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
verisignclass3g5ca	*Apr 21, 2018*	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
autoridaddecertificacionfirmaprofesionalcifa62634068	*Jun 18, 2018*	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
hellenicacademicandresearchinstitutionseccrootca2015	*Jun 18, 2018*	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
verisigntsaca	*Apr 21, 2018*	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
utnuserfirsthardwareca	*Apr 21, 2018*	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7

Name	Date	SHA1 Fingerprint
identrustcommercialrootca1	*Jun 18, 2018*	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
dtrustrootclass3ca22009	*Jun 18, 2018*	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
epkirootcertificationauthority	*Jun 18, 2018*	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert30.pem	*Jun 18, 2018*	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
teliasonerarootcav1	*Jun 18, 2018*	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
buypassclass3ca	*Apr 21, 2018*	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert22.pem	*Jun 18, 2018*	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
mozillacert14.pem	*Jun 18, 2018*	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
verisignc2g2.pem	*Jun 18, 2018*	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
certumca	*Apr 21, 2018*	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert92.pem	*Jun 18, 2018*	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0
mozillacert150.pem	*Jun 18, 2018*	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9

Name	Date	SHA1 Fingerprint
mozillacert84.pem	*Jun 18, 2018*	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2
ttelesecglobalrootclass3	*Jun 18, 2018*	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
globalsignrootca	*Jun 18, 2018*	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
ttelesecglobalrootclass2	*Jun 18, 2018*	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert142.pem	*Jun 18, 2018*	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
mozillacert76.pem	*Jun 18, 2018*	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert134.pem	*Jun 18, 2018*	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
mozillacert68.pem	*Jun 18, 2018*	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
etugracertificationauthority	*Jun 18, 2018*	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert126.pem	*Jun 18, 2018*	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
keynectisrootca	*Apr 21, 2018*	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
mozillacert118.pem	*Jun 18, 2018*	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45

Name	Date	SHA1 Fingerprint
quovadisrootca3	*Jun 18, 2018*	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	*Jun 18, 2018*	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert5.pem	*Jun 18, 2018*	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
verisignc1g3.pem	*Jun 18, 2018*	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
cybertrustglobalroot	*Jun 18, 2018*	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
amzninternalinfosecag3	*Feb 27, 2015*	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
starfieldrootcertificateauthorityg2	*Jun 18, 2018*	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
entrust2048ca	*Apr 21, 2018*	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
swisssignsilvercag2	*Jun 18, 2018*	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
affirmtrustcommercial	*Jun 18, 2018*	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
certinomisrootca	*Jun 18, 2018*	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:88:A8
xrampglobalcaroot	*Jun 18, 2018*	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6

Name	Date	SHA1 Fingerprint
secureglobalca	*Jun 18, 2018*	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
swisssigngoldg2ca	*Apr 21, 2018*	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert21.pem	*Jun 18, 2018*	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
mozillacert13.pem	*Jun 18, 2018*	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
verisignc2g1.pem	*Jun 18, 2018*	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:14:2A
mozillacert91.pem	*Jun 18, 2018*	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
oistewisekeyglobalrootgaca	*Jun 18, 2018*	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
mozillacert83.pem	*Jun 18, 2018*	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:EA:46
entrustevca	*Apr 21, 2018*	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert141.pem	*Jun 18, 2018*	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
mozillacert75.pem	*Jun 18, 2018*	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
mozillacert133.pem	*Jun 18, 2018*	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84

Name	Date	SHA1 Fingerprint
mozillacert67.pem	*Jun 18, 2018*	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
mozillacert125.pem	*Jun 18, 2018*	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert59.pem	*Jun 18, 2018*	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
thawtepremiumserverca	*Apr 21, 2018*	E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:66:66
mozillacert117.pem	*Jun 18, 2018*	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
utnuserfirstclientauthemailca	*Apr 21, 2018*	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
entrustrootcag2	*Apr 21, 2018*	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
mozillacert109.pem	*Jun 18, 2018*	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
digicerttrustedrootg4	*Jun 18, 2018*	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
gdroot-g2.pem	*Jun 18, 2018*	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
comodoaaaservicesroot	*Jun 18, 2018*	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert4.pem	*Jun 18, 2018*	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B

Name	Date	SHA1 Fingerprint
verisignclass3publicprimarycertificationauthorityg5	*Jun 18, 2018*	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
chambersofcommerceroot2008	*Jun 18, 2018*	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
verisignclass3publicprimarycertificationauthorityg4	*Jun 18, 2018*	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3publicprimarycertificationauthorityg3	*Jun 18, 2018*	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
thawtepersonalfreemailca	*Apr 21, 2018*	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
verisignc1g2.pem	*Jun 18, 2018*	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
gtecybertrustglobalca	*Apr 21, 2018*	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
trustcenteruniversalcai	*Apr 21, 2018*	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
camerfirmachamberscommerceca	*Apr 21, 2018*	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
verisignclass1ca	*Apr 21, 2018*	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1

AWS AppSync resolver mapping template changelog

> **Note**
>
> We now primarily support the APPSYNC_JS runtime and its documentation. Please consider using the APPSYNC_JS runtime and its guides here.

Resolver and function mapping templates are versioned. The mapping template version, such as `2018-05-29`) dictates the following:

- The expected shape of the data source request configuration provided by the request template
- The execution behavior of the request mapping template and the response mapping template

Versions are represented using the YYYY-MM-DD format, a later date corresponds to a more recent version. This page lists the differences between the mapping template versions currently supported in AWS AppSync.

Topics

Datasource Operation Availability Per Version Matrix

Operation/Version Supported	2017-02-28	2018-05-29
AWS Lambda Invoke	Yes	Yes
AWS Lambda BatchInvoke	Yes	Yes
None Datasource	Yes	Yes

Operation/Version Supported	2017-02-28	2018-05-29
Amazon OpenSearch GET	Yes	Yes
Amazon OpenSearch POST	Yes	Yes
Amazon OpenSearch PUT	Yes	Yes
Amazon OpenSearch DELETE	Yes	Yes
Amazon OpenSearch GET	Yes	Yes
DynamoDB GetItem	Yes	Yes
DynamoDB Scan	Yes	Yes
DynamoDB Query	Yes	Yes
DynamoDB DeleteItem	Yes	Yes
DynamoDB PutItem	Yes	Yes
DynamoDB BatchGetItem	No	Yes
DynamoDB BatchPutItem	No	Yes
DynamoDB BatchDeleteItem	No	Yes
HTTP	No	Yes
Amazon RDS	No	Yes

Note: Only **2018-05-29** version is currently supported in functions.

Changing the Version on a Unit Resolver Mapping Template

For Unit resolvers, the version is specified as part of the body of the request mapping template. To update the version, simply update the `version` field to the new version.

For example, to update the version on the AWS Lambda template:

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

You need to update the version field from `2017-02-28` to `2018-05-29` as follows:

```
{
    "version": "2018-05-29", ## Note the version
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

Changing the Version on a Function

For functions, the version is specified as the `functionVersion` field on the function object. To update the version, simply update the `functionVersion`. *Note:* Currently, only `2018-05-29` is supported for function.

The following is an example of a CLI command to update an existing function version:

```
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)"
```

Note: It is recommended to omit the version field from the function request mapping template as it will not be honored. If you do specify a version inside a function request mapping template, the version value will be overridden by the value of the `functionVersion` field.

2018-05-29

Behavior Change

- If the datasource invocation result is `null`, the response mapping template is executed.
- If the datasource invocation yields an error, it is now up to you to handle the error, the response mapping template evaluated result will **always** be placed inside the GraphQL response `data` block.

Reasoning

- A `null` invocation result has meaning, and in some application use cases we might want to handle `null` results in a custom way. For example, an application might check if a record exists in an Amazon DynamoDB table to perform some authorization check. In this case, a `null` invocation result would mean the user might not be authorized. Executing the response mapping template now provides the ability to raise an unauthorized error. This behavior provides greater control to the API designer.

Given the following response mapping template:

```
$util.toJson($ctx.result)
```

Previously with `2017-02-28`, if `$ctx.result` came back null, the response mapping template was not executed. With `2018-05-29`, we can now handle this scenario. For example, we can choose to raise an authorization error as follows:

```
# throw an unauthorized error if the result is null
#if ( $util.isNull($ctx.result) )
    $util.unauthorized()
#end
$util.toJson($ctx.result)
```

Note: Errors coming back from a data source are sometimes not fatal or even expected, that is why the response mapping template should be given the flexibility to handle the invocation error and decide whether to ignore it, re-raise it, or throw a different error.

Given the following response mapping template:

```
$util.toJson($ctx.result)
```

Previously, with `2017-02-28`, in case of an invocation error, the response mapping template was evaluated and the result was placed automatically in the `errors` block of the GraphQL response. With `2018-05-29`, we can now choose what to do with the error, re-raise it, raise a different error, or append the error while return data.

Re-raise an Invocation Error

In the following response template, we raise the same error that came back from the data source.

```
#if ( $ctx.error )
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

In case of an invocation error (for example, `$ctx.error` is present) the response looks like the following:

```
{
    "data": {
        "getPost": null
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "DynamoDB:ConditionalCheckFailedException",
            "message": "Conditional check failed exception...",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

Raise a Different Error

In the following response template, we raise our own custom error after processing the error that came back from the data source.

```
#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)
```

In case of an invocation error (for example, `$ctx.error` is present) the response looks like the following:

```
{
    "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
                }
            ]
        }
    ]
}
```

Append an Error to Return Data

In the following response template, we append the same error that came back from the data source while returning data back inside the response. This is also known as a partial response.

```
#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
```

In case of an invocation error (for example, `$ctx.error` is present) the response looks like the following:

```
{
    "data": {
        "getPost": {
            "id": "1",
            "title: "A post"
        }
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "ConditionalCheckFailedException",
            "message": "Conditional check failed exception...",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

Migrating from *2017-02-28* to *2018-05-29*

Migrating from **2017-02-28** to **2018-05-29** is straightforward. Change the version field on the resolver request mapping template or on the function version object. However, note that **2018-05-29** execution behaves differently from **2017-02-28**, changes are outlined here.

Preserving the same execution behavior from *2017-02-28* to *2018-05-29*

In some cases, it is possible to retain the same execution behavior as the **2017-02-28** version while executing a **2018-05-29** versioned template.

Example: DynamoDB PutItem

Given the following **2017-02-28** DynamoDB PutItem request template:

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "attributeValues" : {
        "baz" : ... typed value
    },
    "condition" : {
       ...
    }
}
```

And the following response template:

```
$util.toJson($ctx.result)
```

Migrating to **2018-05-29** changes these templates as follows:

```
{
    "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" : {
       ...
    }
}
```

And changes the response template as follows:

```
## 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)
```

Now that it is your responsibility to handle errors, we chose to raise the same error using `$util.error()` that was returned from DynamoDB. You can adapt this snippet to convert your mapping template to **2018-05-29**, note that if your response template is different you will have to take account of the execution behavior changes.

Example: DynamoDB GetItem

Given the following **2017-02-28** DynamoDB GetItem request template:

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true
```

```
}
```

And the following response template:

```
## map table attribute postId to field Post.id
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))

$util.toJson($ctx.result)
```

Migrating to **2018-05-29** changes these templates as follows:

```
{
    "version" : "2018-05-29", ## Note the new 2018-05-29 version
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true
}
```

And changes the response template as follows:

```
## 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)
```

In the **2017-02-28** version, if the datasource invocation was `null`, meaning there is no item in the DynamoDB table that matches our key, the response mapping template would not execute.

It might be fine for most of the cases, but if you expected the `$ctx.result` to not be `null`, you now have to handle that scenario.

2017-02-28

Characteristics

- If the datasource invocation result is `null`, the response mapping template is **not** executed.
- If the datasource invocation yields an error, the response mapping template is executed and the evaluated result is placed inside the GraphQL response `errors.data` block.

GraphQL type reference

Scalar types in GraphQL represent primitive leaf values in a GraphQL schema. These are the most basic data types that resolve to a single value. Unlike object types, scalar types cannot have sub-fields. GraphQL comes with a set of default scalar types:

- **Int**: A signed 32-bit integer
- **Float**: A signed double-precision floating-point value
- **String**: A UTF-8 character sequence
- **Boolean**: A true or false value
- **ID**: A unique identifier, often used to refetch an object or as the key for a cache

These scalar types serve as the building blocks for more complex types in your schema. They are used to define fields that contain simple, singular values. In addition to these built-in scalars, AWS AppSync provides you with additional scalars for different use cases.

Interfaces and Unions in GraphQL are abstract types that allow for flexible and extensible schema design. They provide mechanisms for grouping related types and enabling polymorphic queries. An Interface in GraphQL is an abstract type that defines a set of fields that a type must include to implement the interface. It serves as a contract for objects by specifying a common set of fields that implementing types must have. Interfaces are useful when you want to return an object or field that can be of several different types, but still have some guaranteed fields. By contrast, a Union in GraphQL represents a type that could be one of several object types, but does not define any common fields between those types. Unions are helpful when you need to return a field that can be of multiple types, and these types don't necessarily share common fields. Both Interfaces and Unions are particularly useful in scenarios where a field might return different types of data, enabling clients to query for specific fields based on the returned type.

This section is used as a reference for schema types.

Topics

Scalar types in GraphQL

A GraphQL object type has a name and fields, and those fields can have sub-fields. Ultimately, an object type's fields must resolve to *scalar* types, which represent the leaves of the query. For more information about object types and scalars, see Schemas and types on the GraphQL website.

In addition to the default set of GraphQL scalars, AWS AppSync also lets you use the **service-defined** scalars that start with the *AWS* prefix. AWS AppSync doesn't support the creation of **user-defined** (custom) scalars. You must use either the default or *AWS* scalars.

You cannot use *AWS* as a prefix for custom object types.

The following section is a reference for schema typing.

Default scalars

GraphQL defines the following default scalars:

Default scalars list

`ID`

A unique identifier for an object. This scalar is serialized like a `String` but isn't meant to be human-readable.

`String`

A UTF-8 character sequence.

`Int`

An integer value between -(2^{31}) and 2^{31}-1.

`Float`

An IEEE 754 floating point value.

`Boolean`

A Boolean value, either `true` or `false`.

AWS AppSync scalars

AWS AppSync defines the following scalars:

AWS AppSync scalars list

`AWSDate`

An extended ISO 8601 date string in the format `YYYY-MM-DD`.

`AWSTime`

An extended ISO 8601 time string in the format `hh:mm:ss.sss`.

`AWSDateTime`

An extended ISO 8601 date and time string in the format `YYYY-MM-DDThh:mm:ss.sssZ`.

> **Note**
>
> The `AWSDate`, `AWSTime`, and `AWSDateTime` scalars can optionally include a time zone offset. For example, the values `1970-01-01Z`, `1970-01-01-07:00`, and `1970-01-01+05:30` are all valid for `AWSDate`. The time zone offset must be either `Z` (UTC) or an offset in hours and minutes (and, optionally, seconds). For example, `±hh:mm:ss`. The seconds field in the time zone offset is considered valid even though it's not part of the ISO 8601 standard.

`AWSTimestamp`

An integer value representing the number of seconds before or after `1970-01-01-T00:00Z`.

`AWSEmail`

An email address in the format `local-part@domain-part` as defined by RFC 822.

`AWSJSON`

A JSON string. Any valid JSON construct is automatically parsed and loaded in the resolver code as maps, lists, or scalar values rather than as the literal input strings. Unquoted strings or otherwise invalid JSON result in a GraphQL validation error.

`AWSPhone`

A phone number. This value is stored as a string. Phone numbers can contain either spaces or hyphens to separate digit groups. Phone numbers without a country code are assumed to be US/North American numbers adhering to the North American Numbering Plan (NANP).

`AWSURL`

A URL as defined by RFC 1738. For example, `https://www.amazon.com/dp/B000NZW3KC/` or `mailto:example@example.com`. URLs must contain a schema (`http`, `mailto`) and can't contain two forward slashes (`//`) in the path part.

`AWSIPAddress`

A valid IPv4 or IPv6 address. IPv4 addresses are expected in quad-dotted notation (`123.12.34.56`). IPv6 addresses are expected in non-bracketed, colon-separated format (`1a2b:3c4b::1234:4567`). You can include an optional CIDR suffix (`123.45.67.89/16`) to indicate subnet mask.

Schema usage example

The following example GraphQL schema uses all of the custom scalars as an "object" and shows the resolver request and response templates for basic put, get, and list operations. Finally, the example shows how you can use this when running queries and mutations.

```
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
}
```

Here's what a request template for `putObject` might look like. A `putObject` uses a `PutItem` operation to create or update an item in your Amazon DynamoDB table. Note that this code snippet doesn't have a configured Amazon DynamoDB table as a data source. This is being used as an example only:

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

The response template for `putObject` returns the results:

```
$util.toJson($ctx.result)
```

Here's what a request template for `getObject` might look like. A `getObject` uses a `GetItem` operation to return a set of attributes for the item given the primary key. Note that this code snippet doesn't have a configured Amazon DynamoDB table as a data source. This is being used as an example only:

```
{
    "version": "2017-02-28",
    "operation": "GetItem",
    "key": {
```

```
        "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
    }
}
```

The response template for `getObject` returns the results:

```
$util.toJson($ctx.result)
```

Here's what a request template for `listObjects` might look like. A `listObjects` uses a `Scan` operation to return one or more items and attributes. Note that this code snippet doesn't have a configured Amazon DynamoDB table as a data source. This is being used as an example only:

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

The response template for `listObjects` returns the results:

```
$util.toJson($ctx.result.items)
```

The following are some examples of using this schema with GraphQL queries:

```
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
    }
}
```

Interfaces and unions in GraphQL

The GraphQL type system supports Interfaces. An interface exposes a certain set of fields that a type must include to implement the interface.

The GraphQL type system also supports Unions. Unions are identical to interfaces, except that they don't define a common set of fields. Unions are generally preferred over interfaces when the possible types do not share a logical hierarchy.

The following section is a reference for schema typing.

Interface examples

We could represent an `Event` interface that represents any kind of activity or gathering of people. Some possible event types are `Concert`, `Conference`, and `Festival`. These types all share common characteristics, including a name, a venue where the event is taking place, and a start and

end date. These types also have differences; a `Conference` offers a list of speakers and workshops, while a `Concert` features a performing band.

In Schema Definition Language (SDL), the `Event` interface is defined as follows:

```
interface Event {
        id: ID!
        name : String!
        startsAt: String
        endsAt: String
        venue: Venue
        minAgeRestriction: Int
}
```

And each of the types implements the `Event` interface as follows:

```
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]
}
```

Interfaces are useful to represent elements that might be of several types. For example, we could search for all events happening at a specific venue. Let's add a `findEventsByVenue` field to the schema as follows:

```
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]
}
```

The `findEventsByVenue` returns a list of `Event`. Because GraphQL interface fields are common to all the implementing types, it's possible to select any fields on the `Event` interface (`id`, `name`, `startsAt`, `endsAt`, `venue`, and `minAgeRestriction`). Additionally, you can access the fields on any implementing type using GraphQL fragments, as long as you specify the type.

Let's examine an example of a GraphQL query that uses the interface.

```
query {
  findEventsAtVenue(venueId: "Madison Square Garden") {
    id
    name
    minAgeRestriction
    startsAt

    ... on Festival {
      performers
    }

    ... on Concert {
      performingBand
    }
```

```
    ... on Conference {
      speakers
      workshops
    }
  }
}
```

The previous query yields a single list of results, and the server could sort the events by start date by default.

```
{
  "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"
        ]
      }
```

```
    ]
  }
}
```

Since results are returned as a single collection of events, using interfaces to represent common characteristics is very helpful for sorting results.

Union examples

As stated earlier, unions don't define common sets of fields. A search result might represent many different types. Using the `Event` schema, you can define a `SearchResult` union as follows:

```
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
```

In this case, to query any field on our `SearchResult` union, you must use fragments:

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
      id
      name
      performers
    }

    ... on Concert {
      id
      name
      performingBand
    }
```

```
    ... on Conference {
      speakers
      workshops
    }
  }
}
```

Type resolution in AWS AppSync

Type resolution is the mechanism by which the GraphQL engine identifies a resolved value as a specific object type.

Going back to the union search example, provided our query yielded results, each item in the results list must present itself as one of the possible types that the `SearchResult` union defined (that is, `Conference`, `Festival`, `Concert`, or `Venue`).

Because the logic to identify a `Festival` from a `Venue` or a `Conference` is dependent on the application requirements, the GraphQL engine must be given a hint to identify our possible types from the raw results.

With AWS AppSync, this hint is represented by a meta field named `__typename`, whose value corresponds to the identified object type name. `__typename` is required for return types that are interfaces or unions.

Type resolution example

Let's reuse the previous schema. You can follow along by navigating to the console and adding the following under the **Schema** page:

```
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
}
```

Let's attach a resolver to the `Query.search` field. In the `Resolvers` section, choose **Attach**, create a new **Data Source** of type *NONE*, and then name it *StubDataSource*. For the sake of this example, we'll pretend we fetched results from an external source, and hard code the fetched results in the request mapping template.

In the request mapping template pane, enter the following:

```
{
    "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"]
        }
    ]
}
```

If the application returns the type name as part of the `id` field, the type resolution logic must parse the `id` field to extract the type name and then add the `__typename` field to each of the results. You can perform that logic in the response mapping template as follows:

> **Note**
>
> You can also perform this task as part of your Lambda function, if you are using the Lambda data source.

```
#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)
```

Run the following query:

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
        id
      name
      performers
    }

    ... on Concert {
      id
      name
      performingBand
    }

    ... on Conference {
      speakers
```

```
      workshops
    }
  }
}
```

The query yields the following results:

```
{
  "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"
        ]
      }
    ]
  }
}
```

The type resolution logic varies depending on the application. For example, you could have a different identifying logic that checks for the existence of certain fields or even a combination of

fields. That is, you could detect the presence of the `performers` field to identify a `Festival` or the combination of the `speakers` and the `workshops` fields to identify a `Conference`. Ultimately, it is up to you to define the logic you want to use.

Troubleshooting and common mistakes in AWS AppSync

This section discusses some common errors and how to troubleshoot them.

Incorrect DynamoDB key mapping

If your GraphQL operation returns the following error message, it may be because your request mapping template structure doesn't match the Amazon DynamoDB key structure:

```
The provided key element does not match the schema (Service: AmazonDynamoDBv2; Status
 Code: 400; Error Code
```

For example, if your DynamoDB table has a hash key called `"id"` and your template says `"PostID"`, as in the following example, this results in the preceding error, because `"id"` doesn't match `"PostID"`.

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "PostID" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

Missing resolver

If you execute a GraphQL operation, such as a query, and get a null response, this may be because you don't have a resolver configured.

For example, if you import a schema that defines a `getCustomer(userId: ID!):` field, and you haven't configured a resolver for this field, then when you execute a query such as `getCustomer(userId:"ID123"){...}`, you'll get a response such as the following:

```
{
    "data": {
    "getCustomer": null
    }
```

```
}
```

Mapping template errors

If your mapping template isn't properly configured, you'll receive a GraphQL response whose `errorType` is `MappingTemplate`. The `message` field should indicate where the problem is in your mapping template.

For example, if you don't have an `operation` field in your request mapping template, or if the `operation` field name is incorrect, you'll get a response like the following:

```
{
    "data": {
        "searchPosts": null
    },
    "errors": [
        {
        "path": [
            "searchPosts"
        ],
        "errorType": "MappingTemplate",
        "locations": [
            {
            "line": 2,
            "column": 3
            }
        ],
        "message": "Value for field '$[operation]' not found."
        }
    ]
}
```

Incorrect return types

The return type from your data source must match the defined type of an object in your schema, otherwise you may see a GraphQL error like:

```
"errors": [
    {
    "path": [
```

```
        "posts"
    ],
    "locations": null,
    "message": "Can't resolve value (/posts) : type mismatch error, expected type LIST,
 got OBJECT"
    }
]
```

For example this could occur with the following query definition:

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

Which expects a LIST of `[Posts]` objects. For example if you had a Lambda function in Node.JS with something like the following:

```
const result = { data: data.Items.map(item => { return item ; }) };
callback(err, result);
```

This would throw an error as `result` is an object. You would need to either change the callback to `result.data` or alter your schema to not return a LIST.

Processing invalid requests

When AWS AppSync is unable to process and send a request (due to improper data such as invalid syntax) to the field resolver, the response payload will return the field data with values set to `null` and any relevant errors.