AWS AppSync

AWS AppSync Developer Guide

AWS AppSync: AWS AppSync Developer Guide

Table of Contents

Welcome

This is prerelease documentation for a service in preview release. It is subject to change.

This is the *AWS AppSync Developer Guide*.

AWS AppSync is an enterprise-level, fully managed GraphQL service with real-time data synchronization and offline programming features.

This guide focuses on using AWS AppSync to create and interact with data sources by using GraphQL from your application. Developers who want to build applications using GraphQL with robust database, search, and compute capabilities, will find the information they need to build an application or integrate existing data sources with AWS AppSync.

Quickstart

This is prerelease documentation for a service in preview release. It is subject to change.

This section describes how to use the AWS AppSync console to launch a sample schema, create and configure a GraphQL API with queries and mutations, and use the API in a sample app.

Alternatively, you can get started with AWS AppSync by writing a GraphQL schema from scratch. For more information, see Designing Your Schema (p. 11).

AWS AppSync also provides a sample application that automatically deploys an API with a GraphQL schema, connects resolvers, and provisions an Amazon DynamoDB data source for you with a single button click. For more information, see Data Sources and Resolvers (p. 80) and Building a Client App (p. 31).

Topics

Launch a Sample Schema

This is prerelease documentation for a service in preview release. It is subject to change.

This section describes how to use the AWS AppSync console to launch a sample schema and create and configure a GraphQL API.

Launch a Sample Schema

The sample schema enables users to create an application where events ("Going to the movies" or "Dinner at Mom & Dad's") can be entered. Application users can also comment on the events ("See you at 7!"). This app demonstrates using GraphQL operations where state is persisted in Amazon DynamoDB.

To start, you'll create a sample schema and provision it.

To create the API

1. Open the AWS AppSync console at https://console.aws.amazon.com/appsync/.
   <step>

   Choose **Create API** from the **Dashboard**.
   </step>
2. Type a friendly application name.
   <step>

   At the bottom of the console window, select the `Sample` schema.
   </step>
3. Choose **Create** and wait for the provisioning process to complete.

Taking a Tour of the Console

After your schema is deployed and your resources are provisioned, you can use the GraphQL API in the AWS AppSync console. The first page you see is **Getting Started**, which has information such as your endpoint URL and authorization mode.

Note: The default authorization mode, API_KEY, uses an API key to test the application. However, for production GraphQL APIs, you should use one of the stronger authorization modes, such as AWS Identity and Access Management with Amazon Cognito identity or user pools. For more information, see Authorization Use Cases (p. 141).

This page has a listing of sample client applications (JavaScript, iOS, etc.) for testing an end-to-end experience. You can clone and download these, as well as the configuration file that has the necessary information (such as your endpoint URL) to get started. Then, follow the instructions on the page to run your app.

Schema Designer

On the left side of the console, choose **Schema** to view the designer. The designer has your sample `Events` schema loaded. Take a look at the schema more closely, and notice that the code editor has linting and error checking capabilities that you can use when you write your own apps.

On the right side of the console are the GraphQL types that have been created, as well as resolvers on different top-level types, such as queries. When adding new types to a schema (for example, `type TODO {...}`), 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 click the **Create Resources** button at the top and select one of these user-defined types from the drop-down menu, you can see how selecting different field options populates a schema design. Don't select anything now, but try this in the future when you design a schema (p. 11).

Resolver Configuration

From the schema designer, choose one of the **resolvers** on the right, next to a field. A new page opens. This page shows the configured data source (with a full listing on the **Data Sources** tab of the console) for a resolver, as well as the associated **Request** and **Response Mapping Template** designers. Sample mapping templates are provided for common use cases. This is also where you can configure custom logic for things such as parsing arguments from a GraphQL request, pagination token responses to clients, and custom query messages to Amazon Elasticsearch Service.

Settings

The **Settings** tab is where you configure things like the authorization method for your API. For more information on these options, see the security overview (p. 141).

Queries

A built-in designer for writing and running GraphQL queries and mutations, including introspection and documentation, is included in the console. We'll cover that next.

Run Queries and Mutations

This is prerelease documentation for a service in preview release. It is subject to change.

In the AWS AppSync console, choose the **Queries** tab on the left to open the GraphQL operations interface. First, note the pane on the right side that enables you to click through the operations, including queries, mutations, and subscriptions that your schema has exposed. Choose the **Mutation** node, and you see a mutation and can add a new event to it: `createEvent(....):Event`. Use this to add something to your database with GraphQL.

Add Data with a GraphQL Mutation

Because there isn't any data yet, the first step is to add some with a GraphQL mutation. You do this with the `mutation` keyword, passing in the appropriate arguments. This works similarly to a function. You can also select which data you want to be returned in the response by putting the fields inside curly braces. Paste the following into the query editor and choose **Run**:

```
mutation {
    createEvent(
        name:"My first GraphQL event"
        where:"Day 1"
        when:"Friday night"
        description:"Catching up with friends"
    ){
        id
        name
        where
        when
        description
    }
}
```

The record is parsed by the GraphQL engine and inserted into your DynamoDB table by a resolver that is connected to a data source. (You can check this in the DynamoDB console.) Notice that you didn't need to pass in an `id`; however, one was generated and returned in the results specified between the curly braces. This is because the sample demonstrates an `autoId()` function in a GraphQL resolver as a best practice for the partition key set on your Amazon DynamoDB resources. For now, just make a note of the returned `id` value for use in the next section.

Retrieve Data with a GraphQL Query

Now that there is a record in your database, running a query returns some results. One of the main advantages of GraphQL is the ability to specify the exact data requirements that your application has in a query. This time, only add a few of the fields inside the curly braces, pass the `id` argument to `getEvent()`, and press the **Run** button at the top:

```
query {
    getEvent(id:"XXXXXX-XXXX-XXXXXXX-XXXX-XXXXXXXXX"){
        name
        where
        description
    }
}
```

This time, only the fields you specified are returned. You can also try listing all events:

```
query getAllEvents {
    listEvents{
        items{
            id
            name
            when
        }
    }
}
```

This time the query shows nested types as well as giving the query a friendly name (`getAllEvents`), which is optional. Experiment by adding or removing and then rerunning the query. When you're done, it's time to connect a client application.

Running an Application

Now that your API is working, you can use a client application to interact with it. AWS AppSync provides samples in several programming languages to get you started. In the AWS AppSync console, at the root of the navigation, select the **name of your API**, and you will see a list of platforms. Clone the appropriate sample to your local workstation, download the configuration file and, if necessary, the GraphQL schema (used on some platforms for code generation). The configuration file contains details, such as the endpoint URL of your GraphQL API and the API key, to include when getting started. You can change this information later when leveraging IAM or Amazon Cognito user pools in production. See Authorization Use Cases (p. 141) for more information.

Next Steps

Now that you've run through the preconfigured schema, you can choose to build an API from scratch, incorporate an existing data source, or build a client application. For more information, see the following sections:

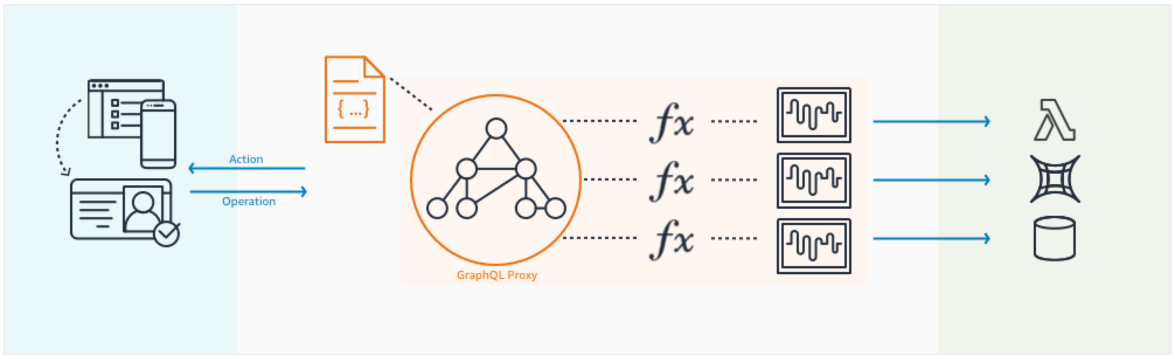
- Designing a GraphQL API (p. 10)
- Connecting Data Sources and Resolvers (p. 80)
- Building Client Applications (p. 31)

System Overview and Architecture

This is prerelease documentation for a service in preview release. It is subject to change.

AWS AppSync allows developers to interact with their data via a managed GraphQL service. GraphQL offers many benefits over traditional gateways, encourages declarative coding style, and works seamlessly with modern tools and frameworks, including React, React Native, iOS, and Android.

Architecture



Concepts

GraphQL Proxy

A component that runs the GraphQL engine for processing requests and mapping them to logical functions for data operations or triggers. The data resolution process performs a batching process (called the Data Loader) to your data sources. This component also manages conflict detection and resolution strategies.

Operation

AWS AppSync supports the three GraphQL operations: query (read-only fetch), mutation (write followed by a fetch), and subscription (long-lived requests that receive data in response to events).

Action

There is one action that AWS AppSync defines. This action is a notification to connected subscribers, which is the result of a mutation. Clients become subscribers through a handshake process following a GraphQL subscription.

Data Source

A persistent storage system or a trigger, along with credentials for accessing that system or trigger. Your application state is managed by the system or trigger defined in a data source.

Resolver

A function that converts the GraphQL payload to the underlying storage system protocol and executes if the caller is authorized to invoke it. Resolvers are comprised of request and response mapping templates, which contain transformation and execution logic.

Identity

A representation of the caller based on a set of credentials, which must be sent along with every request to the GraphQL proxy. It includes permissions to invoke resolvers. Identity information is also passed as context to a resolver and the conflict handler to perform additional checks.

AWS AppSync Client

The location where GraphQL operations are defined. The client performs appropriate authorization wrapping of request statements before submitting to the GraphQL proxy. Responses are persisted in an offline store and mutations are made in a write-through pattern.

GraphQL Overview

This is prerelease documentation for a service in preview release. It is subject to change.

GraphQL is a data language that was developed to enable apps to fetch data from servers. It has a declarative, self-documenting style. In a GraphQL operation, the client specifies how to structure the data when it is returned by the server. This makes it possible for the client to query only for the data it needs, in the format that it needs it in.

GraphQL has three top-level operations:

- Query: read-only fetch
- Mutation: write, followed by a fetch
- Subscription: long-lived connection for receiving data

GraphQL exposes these operations via a schema that defines the capabilities of an API. A schema is comprised of types, which can be root types (query, mutation, or subscription) or user-defined types. Developers start with a schema to define the capabilities of their GraphQL API, which a client application will communicate with. Learn more about this process here (p. 11).

After a schema is defined, the fields on a type need to return some data. The way this happens in a GraphQL API is through a GraphQL resolver. This is a function that either calls out to a data source or invokes a trigger to return some value (such as an individual record or a list of records). Resolvers can have many types of data sources, such as NoSQL databases, relational databases, or search engines. You can aggregate data from multiple data sources and return identical types, mixing and matching to meet your needs.

After a schema is connected to a resolver function, a client app can issue a GraphQL query or, optionally, a mutation or subscription. A query will have the `query` keyword followed by curly braces, and then the field name, such as `allPosts`. After the field name is a second set of curly braces with the data that you want to return. For example:

```
query {
 allPosts {
   id
   author
   title
   content
  }
}
```

This query invokes a resolver function against the `allPosts` field and returns just the `id`, `author`, `title`, and `content` values. If there were many posts in the system (assuming that `allPosts` return blog posts, for example), this would happen in a single network call. Though designs can vary, in traditional systems, this is usually modeled in separate network calls for each post. This reduction in network calls reduces bandwidth requirements and therefore saves battery life and CPU cycles consumed by client applications.

These capabilities make prototyping new applications, and modifying existing applications, very fast. A benefit of this is that the application's data requirements are "co-located" in the application with the UI code for your programming language of choice. This enables client and backend teams to work independently, instead of encoding data modeling on backend implementations.

Finally, the type system provides powerful mechanisms for pagination, relations, inheritance, and interfaces. You can relate different types between separate NoSQL tables when using the GraphQL type system.

For further reading, see the following resources:

- GraphQL
- Designing a GraphQL API (p. 10)
- Data Sources and Resolvers Tutorial (p. 80)

Designing a GraphQL API

This is prerelease documentation for a service in preview release. It is subject to change.

If you are building a GraphQL API, there are some concepts you need to know, such as schema design and how to connect to data sources.

In this section, we describe building a schema from scratch, provisioning resources automatically, manually defining a data source, and connecting to it with a GraphQL resolver. AWS AppSync can also build out a schema and resolvers from scratch, if you have an existing Amazon DynamoDB table.

GraphQL Schema

Each GraphQL API is defined by a single GraphQL schema. 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 referred to as that server's schema. It is made up of a set of object types, scalars, input types, interfaces, enums, and unions. It defines the shape of the data that flows through your API and also the operations that can be performed. GraphQL is a strongly typed protocol and all data operations are validated against this schema.

Data Source

Data sources are resources in your AWS account that GraphQL APIs can interact with. AWS AppSync supports AWS Lambda, Amazon DynamoDB, and Amazon Elasticsearch Service as data sources.

An AWS AppSync API can be configured to interact with multiple data sources, enabling you to aggregate data in a single location.

Resolvers

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

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

AWS Resources

AWS AppSync can use AWS resources from your account that already exist or can provision DynamoDB tables on your behalf from a schema definition.

Topics

Designing Your Schema

This is prerelease documentation for a service in preview release. It is subject to change.

Creating an Empty Schema

Schema files are text files, usually named "schema.graphql". You can create this file and submit it to AWS AppSync by using the CLI or navigating to the console and adding the following under the **Schema** page:

```
schema {
}
```

Every schema has this root for processing. This fails to process until you add a root query type.

Adding a Root Query Type

For this example, we create a `Todo` application. A GraphQL schema must have a root query type, so we add a root type named `Query` with a single `getTodos` field that returns a list containing `Todo` objects. Add the following to your `schema.graphql` file:

```
schema {
    query:Query
}

type Query {
    getTodos: [Todo]
}
```

Notice that we haven't yet defined the `Todo` object type. Let's do that now.

Defining a Todo Type

Now, create a type that contains the data for a `Todo` object:

```
schema {
    query:Query
}

type Query {
    getTodos: [Todo]
}

type Todo {
    id: ID!
    name: String
    description: String
    priority: Int
}
```

Notice that the `Todo` object type has fields that are scalar types, such as strings and integers. Any field that ends in an exclamation point is a required field. The `ID` scalar type is a unique identifier that can be either `String` or `Int`. You can control these in your resolver mapping templates for automatic assignment. You'll see this later.

There are similarities between the `Query` and `Todo` types. In GraphQL, the root types (`Query`, `Mutation`, and `Subscription`) are just types like the ones you define. They're special, though, in that you expose them from your schema as the entry point for your API. For more information, see The Query and Mutation types.

Adding a Mutation Type

Now that you have an object type and can query the data, if you want to add, update, or delete data via the API you need to add a mutation type to your schema. For the Todo example, add this as a field named "addTodo" on a mutation type:

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

type Query {
    getTodos: [Todo]
}

type Mutation {
    addTodo(id: ID!, name: String, description: String, priority: Int): Todo
}

type Todo {
    id: ID!
    name: String
    description: String
    priority: Int
}
```

Notice that mutation is also added to this schema type because it is a root type.

Modifying the Todo with a Status

At this point, your GraphQL API is structurally functioning for reading and writing Todo objects (it just doesn't have a data source, which is described in the next section). You can modify this API with more advanced functionality, such as adding a status to your Todo, which comes from a set of values defined as an ENUM:

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

type Query {
    getTodos: [Todo]
}

type Mutation {
    addTodo(id: ID!, name: String, description: String, priority: Int, status: TodoStatus):
 Todo
}

type Todo {
    id: ID!
    name: String
    description: String
    priority: Int
    status: TodoStatus
}

enum TodoStatus {
    done
    pending
}
```

An ENUM is like a string, but it can take one of a set of values. In the previous example, you added this type, modified the Todo type, and added the Todo field to contain this functionality.

Subscriptions

Real-Time Data (p. 138)

Further Reading

For more information, see the GraphQL type system.

Advanced - Relations and Pagination

Suppose you had a million `todos`. You wouldn't want to fetch all of these every time. Make the following changes to your schema:

- Add a new `TodoConnection` type, which has `todos` and `nextToken` fields.
- Add two input arguments, `first` and `after`, to the `getTodos` field.
- Change `getTodos` so that it returns `TodoConnection`.

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

type Query {
    getTodos(first: Int = 20, after: String): TodoConnection
}

type Mutation {
    addTodo(id: ID!, name: String, description: String, priority: Int, status: TodoStatus):
 Todo
}

type Todo {
    id: ID!
    name: String
    description: String
    priority: Int
    status: TodoStatus
}

type TodoConnection {
    todos: [Todo]
    nextToken: String
}

enum TodoStatus {
    done
    pending
}
```

The `TodoConnection` type allows you to return a list of `todos` and a `nextToken` for getting the next batch of `todos`. In AWS AppSync, this is connected to Amazon DynamoDB with a mapping template. This converts the value of the first argument to the `maxResults` parameter and the `after` argument to the `exclusiveStartKey` parameter. See Resolver Mapping Template Reference (p. 153) for examples.

Next, suppose your todos have comments, and you want to run a query that returns all the comments for a `todo`. Modify your schema to have a `Comment` type, add a `comments` field to the `todo` type, and add an `addComment` field on the `Mutation` type as follows:

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

type Query {
    getTodos(first: Int = 20, after: String): TodoConnection
}

type Mutation {
    addTodo(id: ID!, name: String, description: String, priority: Int, status: TodoStatus):
 Todo
    addComment(todoid: ID!, content: String): Comment
}

type Todo {
    id: ID!
    name: String
    description: String
    priority: Int
    status: TodoStatus
    comments: [Comment]
}

type Comment {
    id: ID!
    content: String
}

type TodoConnection {
    todos: [Todo]
    nextToken: String
}

enum TodoStatus {
    done
    pending
}
```

The application graph on top of your existing data sources in AWS AppSync allows you to return data from two separate data sources in a single GraphQL query. In the example, the assumption is that there is both a Todos table and a Comments table. We'll show how this is done in Configuring Resolvers (p. 22).

Interfaces and Unions in GraphQL

This is prerelease documentation for a service in preview release. It is subject to change.

Interfaces

GraphQL's type system features Interfaces. An interface exposes a certain set of fields that a type must include to implement the interface.

For example, we could represent an `Event` interface that represents any kind of activity or gathering of people. Possible kinds of events are `Concert`, `Conference`, and `Festival`. These types all share common characteristics, they all have 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 SDL, our `Event` interface would be:

```
interface Event {
        id: ID!
        name : String!
        startsAt: String
        endsAt: String
        venue: Venue
        minAgeRestriction: Int
}
```

And each of the types implements the `Event` interface:

```
type Concert implements Event {
    id: ID!
    title: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performingBand: String
}

type Festival implements Event {
    id: ID!
    title: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performers: [String]
}

type Conference implements Event {
    id: ID!
    title: 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 on the schema:

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

```
type Query {
    # Retrieve Events at a specific Venue
    findEventsAtVenue(venueId: ID!): [Event]
}

type Venue {
    id: ID!
    name: String
    address: String
    maxOccupancy: Int
}

type Concert implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performingBand: String
}

interface Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
}

type Festival implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performers: [String]
}

type Conference implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    speakers: [String]
    workshops: [String]
}
```

`findEventsByVenue` returns a list of `Event`. Because GraphQL interface fields are common to all the implementing types, you probably guessed it is possible to select any fields on the `Event` interface (`id`, `title`, `startsAt`, `endsAt`, `venue`, and `minAgeRestriction`). Additionally, we can access the fields on any implementing type, as long as we specify the type, using GraphQL fragments.

Let's look at an example of a GraphQL query that uses our interface.

```
query {
  findEventsAtVenue(venueId: "Madison Square Garden") {
    id
    name
    minAgeRestriction
    startsAt

    ... on Festival {
      performers
    }

    ... on Concert {
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

The previous query would yield a single list of results, and the server could, by default, sort the events by start date.

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

As you can see, results are returned as a single collection of events. Using interfaces to represent common characteristics will be very handy for **sorting** results.

Unions

GraphQL's type system also features Unions. Unions are identical to Interfaces, except they do not define a common set of fields. Unions are generally preferred over Interfaces when the possible types do not share a logical hierarchy.

For example, a search result might represent many different types. Using our `Event` schema, we can define a `SearchResult` union:

```
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, we must use fragments. Let's look at an example:

```
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.

Coming 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 our `SearchResult` union defined. (i.e., `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 our 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 console, select **Attach Resolver**, create a new **Data Source** of type *NONE*, and then name it *StubDataSource*. For the sake of this example, we will pretend we fetched results from an external source, and hardcode the fetched results in our request mapping template.

In the request mapping template pane, enter:

```
{
    "version" : "2017-02-28",
    "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"]
        }
    ]
}
```

In our application, we chose to return the type name as part of the `id` field. Our type resolution logic only consists of parsing the `id` field to extract the type name and adding the `__typename` field to each of the results. We can easily perform that logic in the response mapping template (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)
```

Running our query,

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
        id
      name
      performers
    }

    ... on Concert {
      id
      name
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

will yield the results we specified:

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

Naturally, the type resolution logic will vary depending on the application. For example, we could have a different identifying logic that checks for the existence of certain fields or even a combination of fields. That is, we 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 what the logic will be.

Attaching a Data Source

This is prerelease documentation for a service in preview release. It is subject to change.

(Optional) Automatic Provision

Continuing on from Designing Your Schema (p. 11), you can have AWS AppSync automatically create tables based on your schema definition. You can see that process in (Optional) Provision from Schema (p. 25). You can also skip this and continue on to build from scratch.

Adding a Data Source

Now that you created a schema in the AWS AppSync console and saved it, you can add a data source. The schema in the previous section assumes that you have a Amazon DynamoDB table called "Todos" with a hash key called "id" (and if you're doing the advanced section with Relations, you also need a table named "Comments" with a hash key of "todoid" and a sort key of "content").

To add your data source

1. Choose the **Data Sources** tab in the console, and choose **New**.
   <step>

   Give your data source a friendly name, such as "Todos table".
   </step>
2. Choose **Amazon DynamoDB Table** as the type.
   <step>

   Choose the appropriate region.
   </step>
3. Choose your Todos table. Then either choose an existing role that has IAM permissions for `PutItem` and scan for your table, or create a new role.

If you're doing the advanced section, repeat the process. Note that you need IAM permissions of *PutItem* and *Query* on the "Comments" table.

Now that you've connected a data source to an AWS service, you can connect it to your schema with a resolver. See Configuring Resolvers (p. 22).

Configuring Resolvers

This is prerelease documentation for a service in preview release. It is subject to change.

Create Your First Resolver

Navigate back to the **Schema** page in the AWS AppSync console and find the query type on the right side. Choose the **Attach resolver** button next to the `getTodos` field, which opens the **Add Resolver** page. Select the data source you just created and either use a default template or paste in your own. For common use cases, the AWS AppSync console has built-in templates that you can use for getting items from data sources (all item queries, individual lookups, etc.). For example, on the simple version of the schema from Designing Your Schema (p. 11) where `getTodos` didn't have pagination, the mapping template is as follows:

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

A response mapping template is always needed. The console provides a default with the following passthrough value:

```
$utils.toJson($context.result)
```

Adding a Resolver for Mutations

Repeat the preceding process, starting at the **Schema** page and choosing **Attach resolver** for the `addTodo` mutation. 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" : { "S" : "${context.arguments.id}" }
    },
    "attributeValues" : {
        "name" : { "S" : "${context.arguments.name}" },
        "description" : { "S" : "${context.arguments.description}" },
        "priority" : { "N" : ${context.arguments.priority} },
        "status" : { "S" : "${context.arguments.status}" }
    },
}
```

Notice how the arguments defined in the `addTodo` field from your GraphQL schema are converted into DynamoDB operations.

Use the same passthrough template from earlier.

Advanced Resolvers

If you are following the Advanced section of building a sample schema in Designing Your Schema (p. 11), to do a paginated scan, you should use the following template:

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "nextToken" : "${context.arguments.after}",
    "limit" : ${context.arguments.first}
}
```

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 would be:

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

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.

Note: The fact that this uses a query operation against a second table is only for illustrative purposes. It could also be another operation against DynamoDB. Further, the data could be pulled from another data source, such as AWS Lambda or Amazon Elasticsearch Service because the relation is controlled by your GraphQL schema.

From the **Schema** page in the console, click the comments field on the Todo type, and then choose **Attach resolver**. Use the following request mapping template:

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
     "key": {
        "todoid" : { "S" : "${context.source.id}" }
    },
}
```

Pay attention to "context.source". This references the parent object of the current field being resolved. In this example, "source" is referring to the Todo object, which contains the comments you are fetching.

You can use the passthrough response mapping template.

Finally, create an *addComment* resolver from the schema page in the console, just like you did for the preceding fields. The request mapping template in this case is a simple *PutItem*:

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
     "key": {
        "todoid" : { "S" : "${context.arguments.todoit}" },
        "content" : { "S" : "${context.arguments.content}" }
    }
}
```

In the preceding example, the key corresponds to the arguments from the *addComment* mutation. Use a passthrough response.

Using Your API

This is prerelease documentation for a service in preview release. It is subject to change.

Now that you have a GraphQL API with a schema uploaded, data sources configured, and resolvers connected to your types, you can test your API. Navigate to the **Queries** tab in the console and enter the following text in the editor:

```
mutation add {
    addTodo(id:"123" name:"My TODO" description:"Testing AWS AppSync" priority:2){
        id
        name
        description
        priority
    }
}
```

Press the button at the top to run your mutation. After it completes, the result from your selection set (`id`, `name`, `description`, and `priority`) are displayed on the right. The data is also in the Amazon DynamoDB table for your data source, which you can verify using the console.

Now run a query :

```
query {
    allTodo {
        id
        name
    }
}
```

This should return your data, but just the two fields (`id` and `name`) from your selection set.

(Optional) Provision from Schema

This is prerelease documentation for a service in preview release. It is subject to change.

AWS AppSync can automatically provision Amazon DynamoDB tables from a schema definition, create data sources, and connect the resolvers on your behalf. This can be useful if you want to let AWS AppSync define the appropriate table layout and indexing strategy based on your schema definition and data access patterns.

Schema

These instructions start with the schema outlined in Designing Your Schema (p. 11), as shown next:

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

type Query {
    allTodo: [Todo]
}

type Mutation {
    addTodo(id: ID!, name: String, description: String, priority: Int, status: TodoStatus):
 Todo
}

type Todo {
    id: ID!
    name: String
    description: String
    priority: Int
    status: TodoStatus
}

enum TodoStatus {
    done
    pending
}
```

From the AWS AppSync console, navigate to the **Schema** page, enter the preceding schema into the editor, and choose **Save**.

Provision from Schema

After you save a schema, a **Create resources** button appears in the upper right of the page. Click this to go to the **Create resources** page. You can select any user-defined GraphQL types from the screen, and your `Todo` type should be available. Select this type and you'll see a form you can use to configure the table details. You can change your DynamoDB primary or sort keys here, as well as add additional indexes. At the bottom of the page is a corresponding section for the GraphQL queries and mutations that are then available to you, based on different key selections. AWS AppSync will provision DynamoDB tables that best match your data access pattern for efficient use of your database throughput. An index selection is also available. You can use it for different query options, which set up a DynamoDB Local Secondary Index or Global Secondary Indexes, as appropriate.

For the preceding example schema, you can simply have `id` selected as the primary key and press the **Create** button. After a moment, your DynamoDB tables are created, data sources are created, and resolvers are connected. You can run mutations and queries as described in the Using Your API (p. 24) section. **Note** that there will be a GraphQL `input` type for the arguments of the created schema. For example if you provision from schema with a GraphQL `type Books {...}` then there might be an input type like so:

```
input CreateBooksInput {
    ISBN: String!
    Author: String
    Title: String
    Price: String
}
```

To use this in a GraphQL query or mutation you would do the following:

```
mutation add {
    createBooks(input:{
        ISBN:2349238
        Author:"Nadia Bailey"
        Title:"Running in the park"
        Price:"10"
    }){
        ISBN
        Author
    }
}
```

(Optional) Import from Amazon DynamoDB

This is prerelease documentation for a service in preview release. It is subject to change.

AWS AppSync can automatically create a GraphQL schema and connect resolvers to existing Amazon DynamoDB tables. This can be useful if you have DynamoDB tables for which you want to expose data through a GraphQL endpoint, or if you're more comfortable starting first with your database design instead of a GraphQL schema.

Import a DynamoDB Table

From the AWS AppSync console, navigate to the **Data Sources** page and select the **New** button. Give your data source a friendly name, and select Amazon DynamoDB as the data source type. Select the appropraite table, then toggle the switch under **Automatically generate GraphQL**.

You'll see two code editors with GraphQL schema:

- The top editor can be manipulated to give your type a custom name (such as `type MYNAME {...}`), which will contain the data from your DynamoDB table when you run queries or muations. You can also add fields to the type, such as DynamoDB non-key attributes (which cannot be detected on import).
- The bottom editor is read-only and contains generated GraphQL schema snippets, showing what types, queries, and mutations will be merged into your schema. If you edit the type in the top editor, this will change as appropriate.

Press **Create** at the bottom and your schema is merged and resolvers are created. After this is complete, you can run mutations and queries as described in the Using Your API (p. 24) section. **Note** that there will be a GraphQL `input` type for the arguments of the created schema. For example if you import a table called "Books" then there might be an input type like so:

```
input CreateBooksInput {
    ISBN: String!
    Author: String
    Title: String
    Price: String
}
```

To use this in a GraphQL query or mutation you would do the following:

```
mutation add {
    createBooks(input:{
        ISBN:2349238
        Author:"Nadia Bailey"
        Title:"Running in the park"
        Price:"10"
    }){
        ISBN
        Author
    }
}
```

Example Schema from Import

Suppose that you have a DynamoDB table with the following format:

```
{
    Table: {
        AttributeDefinitions: [
            {
                AttributeName: 'authorId',
                AttributeType: 'S'
            },
            {
                AttributeName: 'bookId',
                AttributeType: 'S'
            },
            {
                AttributeName: 'title',
                AttributeType: 'S'
            }
        ],
        TableName: 'BookTable',
        KeySchema: [
            {
                AttributeName: 'authorId',
                KeyType: 'HASH'
            },
            {
                AttributeName: 'title',
                KeyType: 'RANGE'
            }
        ],
        TableArn: 'arn:aws:dynamodb:us-west-2:012345678910:table/BookTable',
        LocalSecondaryIndexes: [
            {
                IndexName: 'authorId-bookId-index',
                KeySchema: [
                    {
                        AttributeName: 'authorId',
                        KeyType: 'HASH'
                    },
                    {
                        AttributeName: 'bookId',
                        KeyType: 'RANGE'
                    }
                ],
                Projection: {
                    ProjectionType: 'ALL'
                },
                IndexSizeBytes: 0,
                ItemCount: 0,
                IndexArn: 'arn:aws:dynamodb:us-west-2:012345678910:table/BookTable/index/
authorId-bookId-index'
            }
        ],
        GlobalSecondaryIndexes: [
            {
                IndexName: 'title-authorId-index',
                KeySchema: [
                    {
                        AttributeName: 'title',
                        KeyType: 'HASH'
                    },
                    {
                        AttributeName: 'authorId',
                        KeyType: 'RANGE'
                    }
                ],
                Projection: {
                    ProjectionType: 'ALL'
                },
                IndexArn: 'arn:aws:dynamodb:us-west-2:012345678910:table/BookTable/index/
title-authorId-index'
            }
        ]
    }
```

```
}
```

The type editor at the top will show the following:

```
type Book {
    # Key attributes. Changing these may result in unexpected behavior.
    authorId: ID!
    title: String!

    # Index attributes. Changing these may result in unexpected behavior.
    bookId: ID!

    # Add additional non-key attributes below.
    isPublished: Boolean
}
```

This top editor is writable, and the non-key attributes at the bottom like `isPublished` need to be added manually as they cannot be inferred from DynamoDB automatically. For instance if you had another attribute on an item in your DynamoDB table called `rating` you would need to add it under `isPublished` to have it populated in the GraphQL schema. The bottom editor would have the following proposed schema merges:

```
type Query {
    getBook(authorId: ID!, title: String!): Book
    listBooks(first: Int, after: String): BookConnection
    getBookByAuthorIdBookIdIndex(authorId: ID!, bookId: ID!): Book
    queryBooksByAuthorIdBookIdIndex(authorId: ID!, first: Int, after: String):
 BookConnection
    getBookByTitleAuthorIdIndex(title: String!, authorId: ID!): Book
    queryBooksByTitleAuthorIdIndex(title: String!, first: Int, after: String):
 BookConnection
}
type Mutation {
    createBook(input: CreateBookInput!): Book
    updateBook(input: UpdateBookInput!): Book
    deleteBook(input: DeleteBookInput!): Book
}
type Subscription {
    onCreateBook(authorId: ID, title: String, bookId: ID, isPublished: Boolean): Book
 @aws_subscribe(mutations: ["createBook"])
    onUpdateBook(authorId: ID, title: String, bookId: ID, isPublished: Boolean): Book
 @aws_subscribe(mutations: ["updateBook"])
    onDeleteBook(authorId: ID, title: String, bookId: ID, isPublished: Boolean): Book
 @aws_subscribe(mutations: ["deleteBook"])
}
input CreateBookInput {
    authorId: ID!
    title: String!
    bookId: ID!
    isPublished: Boolean
}
input UpdateBookInput {
    authorId: ID!
    title: String!
    bookId: ID
    isPublished: Boolean
}
input DeleteBookInput {
    authorId: ID!
    title: String!
}
type BookConnection {
    items: [Book]
    nextToken: String
}
```

Building a Client App

This is prerelease documentation for a service in preview release. It is subject to change.

The following sections are tutorials for building a client application with GraphQL on different platforms. Each tutorial starts with an application running with local data, and then adds in the AWS AppSync SDK to communicate with your GraphQL API. The tutorials assume you have a basic schema created using the schema from the DynamoDB resolvers tutorial (p. 80) as a reference starting point, which you can optionally complete first.

Topics

Building a ReactJS Client App

This is prerelease documentation for a service in preview release. It is subject to change.

AWS AppSync integrates with the Apollo GraphQL client for building client applications. AWS provides Apollo plugins for offline support, authorization, and subscription handshaking. You can use the Apollo client directly, or you can use it with some of the client helpers provided in the AWS AppSync SDK. This tutorial shows you how to use AWS AppSync with React Apollo, which uses ReactJS constructs and patterns with GraphQL.

Before You Begin

This tutorial is set up for a sample API using the schema from the DynamoDB resolvers tutorial (p. 80). To follow along with the complete flow, you can optionally walk through that tutorial first. If you want to do more customization of GraphQL resolvers, such as those that use DynamoDB, see the Resolver Mapping Template Reference (p. 153). The application will use the following starting schema:

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

type Mutation {
    addPost(id: ID! author: String! title: String content: String! url: String!): Post!
    updatePost(id: ID! author: String! title: String content: String url: expectedVersion:
 Int!): Post!
```

```
    deletePost(id: ID!, expectedVersion: Int): Post
}

type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
}

type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}

type Query {
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID!): Post
}
```

This schema defines a `Post` type and operations to add, get, update, and delete `Post` objects.

Get the GraphQL API Endpoint

After you create your GraphQL API, you'll need to get the API endpoint (URL) so you can use it in your client application. You can get the API endpoint in two ways.

In the AWS AppSync console, choose **Home** and then choose **GraphQL URL** to see the API endpoint.

Alternatively, you can get it by running the following CLI command:

```
aws appsync get-graphql-api --api-id $GRAPHQL_API_ID
```

The following instructions show how you can use `AWS_IAM` for client authorization. In the AWS AppSync console, choose **Settings** on the left, and then choose **AWS_IAM**. For more information about authorization modes, see Authorization Use Cases (p. 141).

Download a Client Application

To show you how to use AWS AppSync, we first review a React application with just a local array of data. Then we add AWS AppSync capabilities to it. To begin, download a sample application where we can add, update, and delete posts.

Understanding the React Sample App

The React sample app has three major files:

- `./src/App.js`: The main entry point of the application. It renders the main application shell with two components named `AddPost` and `AllPosts`, and has a local array of data named `posts` which is passed as a prop to the other components.
- `./src/Components/AddPost`: A React component that contains a form that enables a user to enter new information about a post, such as the author and title.

- `./src/Components/AllPosts`: A React component that lists all existing posts from the `posts` array that `App.js` created. It enables you to edit or delete existing posts.

Run your app as follows, and test it to be sure it works:

```
yarn && yarn start
```

Import the AWS AppSync SDK into Your App

In this section, you'll add AWS AppSync to your existing app.

Add the following dependencies to your application:

```
yarn add react-apollo graphql-tag aws-sdk
```

Next, add in the AWS AppSync SDK, including the React extensions:

```
yarn add aws-appsync
yarn add aws-appsync-react
```

From the AWS AppSync console, navigate to your GraphQL API landing page where the **API URL** is listed. At the bottom of the page, choose **Web**. Next, click the **Download** button and save the **AppSync.js** configuration file into `./src`.

To interact with AWS AppSync, your client needs to define GraphQL queries and mutations. This is commonly done in separate files, as follows:

```
mkdir ./src/Queries
touch ./src/Queries/AllPostsQuery.js
touch ./src/Queries/DeletePostMutation.js
touch ./src/Queries/NewPostMutation.js
touch ./src/Queries/UpdatePostMutation.js
```

Edit and save `AllPostsQuery.js`:

```
import gql from 'graphql-tag';

export default gql`
query AllPosts {
    allPost {
        posts {
            __typename
            id
            title
            author
            version
        }
    }
}`;
```

Edit and save `DeletePostMutation.js`:

```
import gql from 'graphql-tag';
```

```
export default gql`
mutation DeletePostMutation($id: ID!, $expectedVersion: Int!) {
    deletePost(id: $id, expectedVersion: $expectedVersion) {
        __typename
        id
        author
        title
        version
    }
}`;
```

Edit and save `NewPostMutation.js`:

```
import gql from 'graphql-tag';

export default gql`
mutation AddPostMutation($id: ID!, $author: String!, $title: String!) {
    addPost(
        id: $id
        author: $author
        title: $title
        content: " "
        url: " "
    ) {
        __typename
        id
        author
        title
        version
    }
}`;
```

Edit and save `UpdatePostMutation.js`:

```
import gql from 'graphql-tag';

export default gql`
mutation UpdatePostMutation($id: ID!, $author: String, $title: String, $expectedVersion:
 Int!) {
    updatePost(
        id: $id
        author: $author
        title: $title
        expectedVersion: $expectedVersion
    ) {
        __typename
        id
        author
        title
        version
    }
}`;
```

Edit your `App.js` file, as follows:

```
import AWSAppSyncClient from "aws-appsync";
import { Rehydrated } from 'aws-appsync-react';
import { AUTH_TYPE } from "aws-appsync/lib/link/auth-link";
import { graphql, ApolloProvider, compose } from 'react-apollo';
import * as AWS from 'aws-sdk';
import AppSync from './AppSync.js';
import AllPostsQuery from './Queries/AllPostsQuery';
import NewPostMutation from './Queries/NewPostMutation';
import DeletePostMutation from './Queries/DeletePostMutation';
import UpdatePostMutation from './Queries/UpdatePostMutation';
```

After all the `import` statements, add the following code:

```
const client = new AWSAppSyncClient({
    url: AppSync.graphqlEndpoint,
    region: AppSync.region,
    auth: {
        type: AUTH_TYPE.API_KEY,
        apiKey: AppSync.apiKey,

        // type: AUTH_TYPE.AWS_IAM,
        // Note - Testing purposes only
        /*credentials: new AWS.Credentials({
            accessKeyId: AWS_ACCESS_KEY_ID,
            secretAccessKey: AWS_SECRET_ACCESS_KEY
        })*/

        // Amazon Cognito Federated Identities using AWS Amplify
        //credentials: () => Auth.currentCredentials(),

        // Amazon Cognito user pools using AWS Amplify
        // type: AUTH_TYPE.AMAZON_COGNITO_USER_POOLS,
        // jwtToken: async () => (await Auth.currentSession()).getIdToken().getJwtToken(),
    },
});
```

You can switch the **AUTH_TYPE** value use API keys, IAM (including short-term credentials from Amazon Cognito Federated Identities), or Amazon Cognito user pools. We recommend you use either IAM or Amazon Cognito user pools after onboarding with an API key. The previous code shows how to use the default configuration of AWS AppSync with an API key, referencing the `AppSync.js` file you downloaded. When you're ready to add other authorization methods to your application, you can use the AWS Amplify library to quickly add these capabilities to your application. The corresponding AWS Amplify methods for the AWS AppSync client constructor are included above. An import of the library with configuration would look similar to the following:

```
import Amplify, { Auth } from 'aws-amplify';
import { withAuthenticator } from 'aws-amplify-react';
Amplify.configure(awsmobile);

//...code

const AppWithAuth = withAuthenticator(App, true);
```

For more information on using AWS Amplify, see the library documentation.

Replace the `App` component entirely, so it looks like this:

```
class App extends Component {
    render() {
        return (
        <div className="App">
            <header className="App-header">
                <img src={logo} className="App-logo" alt="logo" />
                <h1 className="App-title">Welcome to React</h1>
            </header>
            <p className="App-intro">
                To get started, edit <code>src/App.js</code> and save to reload.
            </p>
            <NewPostWithData />
            <AllPostsWithData />
        </div>
        );
    }
}
```

You can also delete the `posts` variable in your code, because the app state will be coming from AWS AppSync.

At the bottom of your `App.js` file, define the following higher-order component (HOC):

```
const AllPostsWithData = compose(
    graphql(AllPostsQuery, {
        options: {
            fetchPolicy: 'cache-and-network'
        },
        props: (props) => ({
            posts: props.data.allPost && props.data.allPost.posts,
        })
    }),
    graphql(DeletePostMutation, {
        props: (props) => ({
            onDelete: (post) => props.mutate({
                variables: { id: post.id, expectedVersion: post.version },
                optimisticResponse: () => ({ deletePost: { ...post, __typename:
 'Post' } }),
            })
        }),
        options: {
            refetchQueries: [{ query: AllPostsQuery }],
            update: (proxy, { data: { deletePost: { id } } }) => {
                const query = AllPostsQuery;
                const data = proxy.readQuery({ query });

                data.allPost.posts = data.allPost.posts.filter(post => post.id !== id);

                proxy.writeQuery({ query, data });
            }
        }
    }),
    graphql(UpdatePostMutation, {
        props: (props) => ({
            onEdit: (post) => {
                props.mutate({
                variables: { ...post, expectedVersion: post.version },
                optimisticResponse: () => ({ updatePost: { ...post, __typename: 'Post',
 version: post.version + 1 } }),
                })
            }
        }),
        options: {
            refetchQueries: [{ query: AllPostsQuery }],
            update: (dataProxy, { data: { updatePost } }) => {
                const query = AllPostsQuery;
                const data = dataProxy.readQuery({ query });

                data.allPost.posts = data.allPost.posts.map(post => post.id !==
 updatePost.id ? post : { ...updatePost });

                dataProxy.writeQuery({ query, data });
            }
        }
    })
    )(AllPosts);

    const NewPostWithData = graphql(NewPostMutation, {
    props: (props) => ({
        onAdd: post => props.mutate({
            variables: post,
            optimisticResponse: () => ({ addPost: { ...post, __typename: 'Post', version:
 1 } }),
        })
    }),
    options: {
        refetchQueries: [{ query: AllPostsQuery }],
        update: (dataProxy, { data: { addPost } }) => {
            const query = AllPostsQuery;
            const data = dataProxy.readQuery({ query });

            data.allPost.posts.push(addPost);

            dataProxy.writeQuery({ query, data });
        }
    }
})(AddPost);
```

Finally, replace `export default App` with the `ApolloProvider`:

```
const WithProvider = () => (
    <ApolloProvider client={client}>
        <Rehydrated>
            <App />
        </Rehydrated>
    </ApolloProvider>
);

export default WithProvider;
```

Test Your Application

```
yarn start
```

Open a webpage and add, remove, edit, and delete data. If you're using Chrome developer tools, you can use the network conditioning tool for offline testing.

Offline Settings

There are important considerations that you'll need to account for if you want an optimistic UI for an application, where data can be manipulated when the device is in an offline state. Many of these settings are documented in the official Apollo documentation, however, we call out several of them here that you should configure.

First, know that the AWS AppSync client allows you to disable offline capabilities if you simply want to use GraphQL in an always-online scenario. To do this, you pass an additional option when instantiating your client, named `disableOffline`, as follows:

```
const client = new AWSAppSyncClient({
    url: AppSync.graphqlEndpoint,
    region: AppSync.region,
    auth: {
        type: AUTH_TYPE.API_KEY,
        apiKey: AppSync.apiKey,
    },
    disableOffline: true
});
```

- fetchPolicy: This option allows you to specify how a query interacts with the network versus local in-memory caching. AWS AppSync persists this cache to a platform-specific storage medium. If you are using the AWS AppSync client in offline scenarios (`disableOffline:false`), you **MUST** set this value to `cache-and-network`:

```
options: {
  fetchPolicy: 'cache-and-network'
}
```

- optimisticResponse: This option allows you to pass a function or an object to a mutation for updating your UI before the server responds with the result. This is needed in offline scenarios (and for slower networks) to ensure that the UI is updated when the device has no connectivity. Optionally, you can also use this if you have set `disableOffline:true`. For example, if you were adding a new object to a list, you might use the following:

```
onAdd: post => props.mutate({
  variables: post,
  optimisticResponse: () => ({ addPost: { __typename: 'Post', ups: 1, downs: 1, content:
 '', url: '', version: 1, ...post } }),
})
```

Normally, you use `optimisticResponse` in conjunction with the `update` option for React Apollo's component, which can trigger during an offline mutation. If you want the UI to update offline for a specific query, you need to specify that query as part of the `readQuery` and `writeQuery` options on the cache, as shown below:

```
options: {
    refetchQueries: [{ query: AllPostsQuery }],
    update: (dataProxy, { data: { addPost } }) => {
        const query = AllPostsQuery;
        const data = dataProxy.readQuery({ query });
        data.allPost.posts.push(addPost);
        dataProxy.writeQuery({ query, data });
      }
   }
```

When this happens, the AWS AppSync persistent store update automatically in response to the Apollo cache update. Upon network reconnection, it will synchronize with your GraphQL endpoint. You could also modify more than one query when offline, in which case you could run the above process multiple times in the same `update` block.

Make Your Application Real Time

Edit your schema with the subscription type, as follows:

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

type Mutation {
    addPost(id: ID! author: String! title: String content: String! url: String!): Post!
    updatePost(id: ID! author: String! title: String content: String url: expectedVersion:
 Int!): Post!
    deletePost(id: ID!, expectedVersion: Int): Post
}

type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
}

type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}

type Query {
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID!): Post
}

type Subscription {
    newPost: Post
    @aws_subscribe(mutations:["addPost"])
}
```

Notice that the `@aws_subscribe` specifies which mutations trigger a subscription. You can add more mutations in this array to meet your application needs.

The subscription type `newPost` needs to be passed into an option (named `updateQuery`) of the React Apollo client to update your UI dynamically when a subcription is received. Ensure that this field name matches the subscription type in the following example code.

In your `App.js` file, edit the `AllPostsWithData` HOC to include `subscribeToNewPost` in the `props` field, as follows:

```
const AllPostsWithData = compose(
    graphql(AllPostsQuery, {
        options: {
            fetchPolicy: 'cache-and-network'
        },
        props: (props) => ({
            posts: props.data.allPost && props.data.allPost.posts,
            // START - NEW PROP :
            subscribeToNewPosts: params => {
                props.data.subscribeToMore({
                    document: NewPostsSubscription,
                    updateQuery: (prev, { subscriptionData: { data : { newPost } } }) => ({
                        ...prev,
                        allPost: { posts: [newPost, ...prev.allPost.posts.filter(post =>
 post.id !== newPost.id)], __typename: 'PaginatedPosts' }
                    })
                });
            },
            // END - NEW PROP
        }
    })
}),
...//more code
```

```
touch src/Queries/NewPostsSubscription.js
```

```
import gql from 'graphql-tag';

export default gql`
    subscription NewPostSub {
    newPost {
        __typename
        id
        title
        author
        version
    }
}`;
```

Add the following `import` statement at the top of your `App.js` file:

```
import NewPostsSubscription from './Queries/NewPostsSubscription';
```

Add the following lifecycle method to your `AllPosts` component in `AllPosts.jsx`:

```
componentWillMount(){
    this.props.subscribeToNewPosts();
}
```

Now try running your app again by typing `yarn start`. Add a new post via the console, with a mutation on `addPost`. You should see real-time data appear in your client application.

Complex Objects

Many times you might want to create logical objects that have more complex data, such as images or videos, as part of their structure. For instance you might create a "Person" type with a profile picture or a "Post" type that has an associated image. With AWS AppSync, you can model these as GraphQL types. If any of your mutations have a variable with `bucket`, `key`, `region`, `mimeType` and `localUri` fields, the SDK will upload the file to Amazon S3 for you.

Edit your schema to add the `S3Object` and `S3ObjectInput` types, as follows:

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

type Mutation {
    addPost(id: ID! author: String! title: String content: String! url: String! file:
 S3ObjectInput): Post!
    updatePost(id: ID! author: String! title: String content: String url: expectedVersion:
 Int!): Post!
    deletePost(id: ID!, expectedVersion: Int): Post
}

type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    file: S3Object
    version: Int!
}

type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}

type S3Object {
    bucket: String!
    key: String!
    region: String!
}

input S3ObjectInput {
    bucket: String!
    key: String!
    region: String!
    localUri: String
    mimeType: String
}

type Query {
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID!): Post
}

type Subscription {
    newPost: Post
    @aws_subscribe(mutations:["addPost"])
}
```

Edit your `./src/Components/AddPost.jsx` file, as follows:

```
import React, { Component } from "react";
import { v4 as uuid } from 'uuid';

export default class AddPost extends Component {

    constructor(props) {
        super(props);
        this.state = this.getInitialState();
    }

    static defaultProps = {
        onAdd: () => null
    }

    getInitialState = () => ({
        id: '',
        title: '',
        author: '',
        file: null,
    });

    handleChange = (field, event) => {
        const { target: { value } } = event;

        this.setState({
            [field]: value
        });
    }

    handleAdd = () => {
        const { title, author, file: selectedFile } = this.state;

        let file;

        if (selectedFile) {
            const { name, type: mimeType } = selectedFile;
            const [, , , extension] = /([^.]+)(\.(\w+))?$/.exec(name);

            const bucket = '[YOUR BUCKET]';
            const key = [uuid(), extension].filter(x => !!x).join('.');
            const region = '[YOUR REGION]';

            file = {
                bucket,
                key,
                region,
                mimeType,
                localUri: selectedFile,
            };
        }

        this.setState(this.getInitialState(), () => {
            this.props.onAdd({ title, author, content: 'hardcoded', file });
        });
    }

    handleCancel = () => {
        this.setState(this.getInitialState());
    }

    render() {
        return (
            <fieldset >
                <legend>Add new Post</legend>
                <div>
                    <label>ID<input type="text" placeholder="ID" value={this.state.id}
 onChange={this.handleChange.bind(this, 'id')} /></label>
                </div>
                <div>
                    <label>Title<input type="text" placeholder="Title"
 value={this.state.title} onChange={this.handleChange.bind(this, 'title')} /></label>
                </div>
                <div>
                    <label>Author<input type="text" placeholder="Author"
 value={this.state.author} onChange={this.handleChange.bind(this, 'author')} /></label>
                </div>
                 <div>
                    <label>File<input type="file" onChange={this.handleChange.bind(this,
 'file')} /></label>
                </div>
                <div>
                    <button onClick={this.handleAdd}>Add new post</button>
                    <button onClick={this.handleCancel}>Cancel</button>
                </div>
            </fieldset>
        );
    }
}
```

Now try running your app again by typing `yarn start`. Add a new post via the console, with a mutation on `addPost`. Your file should be uploaded to Amazon S3 before doing your mutation.

Conflict Resolution

When clients make a mutation, either online or offline, they can send a version number with the payload (named `expectedVersion`) for AWS AppSync to check before writing to Amazon DynamoDB. A DynamoDB resolver mapping template can be configured to perform conflict resolution in the cloud, which you can learn about in Resolver Mapping Template Reference for DynamoDB (p. 166). If the service determines it needs to reject the mutation, data is sent to the client and you can optionally run an additional callback to perform client-side conflict resolution.

For example, suppose you had a mutation with DynamoDB set for checking the version, and the client sent `expectedVersion:0`, as in this example:

```
graphql(UpdatePostMutation, {
        props: (props) => ({
        onEdit: (post) => {
            props.mutate({
                variables: { ...post, expectedVersion: 0 },
                optimisticResponse: () => ({ updatePost: { ...post, __typename: 'Post',
 version: post.version + 1 } }),
            })
        }
    }),...more code
```

This would fail the version check because **0** would be lower than any of the current values. You can then define a custom callback conflict resolver. A custom conflict resolver will receive the following variables:

- mutation: GraphQL statement of a mutation
- mutationName: Optional if a name of a mutation is set on a GraphQL statement
- variables: Input parameters of the mutation
- data: Response from AWS AppSync of actual data in DynamoDB
- retries: Number of times a mutation has been retried

For example, you could have the following custom callback conflict resolver:

```
const conflictResolver = ({ mutation, mutationName, variables, data, retries }) => {
    switch (mutationName) {
        case 'UpdatePostMutation':
            return {
                ...variables,
                expectedVersion: data.version,
            };
        default:
            return false;
    }
}
```

In the example above, you can do a logical check on the `mutationName` and then rerun the mutation with the correct version that AWS AppSync returned.

Note: We recommend doing this only in rare cases. Usually, you should let the AWS AppSync service define conflict resolution, or race conditions can occur. If you don't want to retry, simply return `DISCARD`.

Now, to use this callback, pass it into the AWS AppSync client instantiation:

```
const client = new AWSAppSyncClient({
  url: awsconfig.ENDPOINT,
  region: awsconfig.REGION,
  auth: authInfo,
  conflictResolver,
});
```

Building a React Native Client App

This is prerelease documentation for a service in preview release. It is subject to change.

AWS AppSync integrates with the Apollo GraphQL client for building client applications. AWS provides Apollo plugins for offline support, authorization, and subscription handshaking. You can use the Apollo client directly, or you can use it with some of the client helpers provided in the AWS AppSync SDK. This tutorial shows you how to use AWS AppSync with React Apollo, which uses ReactJS constructs and patterns with GraphQL.

Before You Begin

This tutorial is set up for a sample API using the schema from the DynamoDB resolvers tutorial (p. 80). To follow along with the complete flow, you can optionally walk through that tutorial first. If you want to do more customization of GraphQL resolvers, such as those that use DynamoDB, see the Resolver Mapping Template Reference (p. 153). The application will use the followibng starting schema:

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

type Mutation {
    addPost(id: ID! author: String! title: String content: String! url: String!): Post!
    updatePost(id: ID! author: String! title: String content: String url: expectedVersion:
 Int!): Post!
    deletePost(id: ID!, expectedVersion: Int): Post
}

type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
}

type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}

type Query {
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID!): Post
}
```

This schema defines a `Post` type and operations to add, get, update, and delete `Post` objects.

Get the GraphQL API Endpoint

After you create your GraphQL API, you'll need to get the API endpoint (URL) so you can use it in your client application. There are two ways to get the API endpoint.

In the AWS AppSync console, choose **Home**, and then choose **GraphQL URL** to see the API endpoint.

Alternatively, you can get it by running the following CLI command:

```
aws appsync get-graphql-api --api-id $GRAPHQL_API_ID
```

Download a Client Application

To show you how to use AWS AppSync, we first review a React Native application (bootstrapped with create-react-native-app) with just a local array of data. Then we add AWS AppSync capabilities to it. To begin, download a sample application where we can add, update, and delete posts.

Understanding the React Native Sample App

The React Native sample app has three major files:

- `./src/App.js`: The main entry point of the application. Renders the main application shell with two components named `AddPost` and `AllPosts`, and has a local array of data named `posts` that is passed as a prop to the other components.
- `./src/Components/AddPost`: A React Native component that contains a form that enables a user to enter new information about a post, such as the author and title.
- `./src/Components/AllPosts`: A React Native component that lists all existing posts from the `posts` array that `App.js` created. It enables you to edit or delete existing posts.

Run your app as follows, and test it to be sure it works:

```
yarn && yarn start
```

Import the AWS AppSync SDK into Your App

In this section, you'll add AWS AppSync to your existing React Native app.

For Android, you need to eject and add a permission to access network state. First, run the following:

```
yarn eject
```

After ejecting, edit `android/app/src/main/AndroidManifest.xml` with the following:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.samplereactnative"
    android:versionCode="1"
    android:versionName="1.0">

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Add the following dependencies to your application:

```
yarn add react-apollo graphql-tag aws-sdk
```

Next, add in the AWS AppSync SDK, including the React extensions:

```
yarn add aws-appsync
yarn add aws-appsync-react
```

From the AWS AppSync console, navigate to your GraphQL API landing page where the **API URL** is listed. At the bottom of the page, choose **Web**. Next, click the **Download** button and save the **AppSync.js** configuration file into `./`.

To interact with AWS AppSync, your client needs to define GraphQL queries and mutations. This is commonly done in separate files, as follows:

```
mkdir ./Queries
touch ./Queries/AllPostsQuery.js
touch ./Queries/DeletePostMutation.js
touch ./Queries/NewPostMutation.js
touch ./Queries/UpdatePostMutation.js
```

Edit and save `AllPostsQuery.js`:

```
import gql from 'graphql-tag';

export default gql`
query AllPosts {
    allPost {
        posts {
            __typename
            id
            title
            author
            version
        }
    }
}`;
```

Edit and save `DeletePostMutation.js`:

```
import gql from 'graphql-tag';

export default gql`
mutation DeletePostMutation($id: ID!, $expectedVersion: Int!) {
    deletePost(id: $id, expectedVersion: $expectedVersion) {
        __typename
        id
        author
        title
        version
    }
}`;
```

Edit and save `NewPostMutation.js`:

```
import gql from 'graphql-tag';

export default gql`
mutation AddPostMutation($id: ID!, $author: String!, $title: String!) {
    addPost(
        id: $id
        author: $author
        title: $title
        content: " "
        url: " "
    ) {
        __typename
        id
        author
        title
        version
    }
}`;
```

Edit and save `UpdatePostMutation.js`:

```
import gql from 'graphql-tag';

export default gql`
mutation UpdatePostMutation($id: ID!, $author: String, $title: String, $expectedVersion:
 Int!) {
    updatePost(
        id: $id
        author: $author
        title: $title
        expectedVersion: $expectedVersion
    ) {
        __typename
        id
        author
        title
        version
    }
}`;
```

Edit your `App.js` file, as follows:

```
import AWSAppSyncClient from "aws-appsync";
import { Rehydrated } from 'aws-appsync-react';
import { AUTH_TYPE } from "aws-appsync/lib/link/auth-link";
import { graphql, ApolloProvider, compose } from 'react-apollo';
import * as AWS from 'aws-sdk';
import AppSync from './AppSync.js';
import AllPostsQuery from './Queries/AllPostsQuery';
import NewPostMutation from './Queries/NewPostMutation';
import DeletePostMutation from './Queries/DeletePostMutation';
import UpdatePostMutation from './Queries/UpdatePostMutation';
```

After all the `import` statements, add the following code:

```
const client = new AWSAppSyncClient({
    url: AppSync.graphqlEndpoint,
    region: AppSync.region,
    auth: {
        type: AUTH_TYPE.API_KEY,
        apiKey: AppSync.apiKey,

        //type: AUTH_TYPE.AWS_IAM,
        //Note - Testing purposes only
        /*credentials: new AWS.Credentials({
            accessKeyId: AWS_ACCESS_KEY_ID,
            secretAccessKey: AWS_SECRET_ACCESS_KEY
        })*/

        //IAM Cognito Identity using AWS Amplify
        //credentials: () => Auth.currentCredentials(),

        //Cognito User Pools using AWS Amplify
        // type: AUTH_TYPE.AMAZON_COGNITO_USER_POOLS,
        // jwtToken: async () => (await Auth.currentSession()).getIdToken().getJwtToken(),
    },
});
```

Note that you can switch the **AUTH_TYPE** value to use API keys, IAM (including short-term credentials from Amazon Cognito Federated Identities), or Amazon Cognito user pools. We recommend you use either IAM or Amazon Cognito user pools after onboarding with an API key. The previous code shows how to use the default configuration of AWS AppSync with an API key, referencing the `AppSync.js` file you downloaded. When you're ready to add other authorization methods to your application, you can use the AWS Amplify library to quickly add these capabilities to your application. The corresponding AWS Amplify methods for the AWS AppSync client constructor are included above, and an import of the library with configuration would look similar to the following:

```
import Amplify, { Auth } from 'aws-amplify';
import { withAuthenticator } from 'aws-amplify-react';
Amplify.configure(awsmobile);

//...code

const AppWithAuth = withAuthenticator(App, true);
```

For more information on using AWS Amplify, see the library documentation.

Replace the `App` component entirely, so it looks like this:

```
class App extends Component {
    state = { posts: [] };

    render() {
        return (
        <View style={styles.container}>
            <AddPostWithData />
            <AllPostsWithData />
        </View>
        );
    }
}
```

Delete the `posts` variable in your code, because the app state will be coming from AWS AppSync. Also, change the initial state of the App component to this:

```
state = { posts: [] };
```

At the bottom of your `App.js` file, define the following higher-order component (HOC):

```
const AllPostsWithData = compose(
    graphql(AllPostsQuery, {
        options: {
            fetchPolicy: 'cache-and-network'
        },
        props: (props) => ({
            posts: props.data.allPost && props.data.allPost.posts,
        })
    }),
    graphql(DeletePostMutation, {
        props: (props) => ({
            onDelete: (post) => props.mutate({
                variables: { id: post.id, expectedVersion: post.version },
                optimisticResponse: () => ({ deletePost: { ...post, __typename:
'Post' } }),
            })
        }),
        options: {
            refetchQueries: [{ query: AllPostsQuery }],
            update: (proxy, { data: { deletePost: { id } } }) => {
                const query = AllPostsQuery;
                const data = proxy.readQuery({ query });

                data.allPost.posts = data.allPost.posts.filter(post => post.id !== id);

                proxy.writeQuery({ query, data });
            }
        }
    }),
    graphql(UpdatePostMutation, {
        props: (props) => ({
        onEdit: (post) => {
            props.mutate({
                variables: { ...post, expectedVersion: post.version },
                optimisticResponse: () => ({ updatePost: { ...post, __typename: 'Post',
version: post.version + 1 } }),
            })
        }
        }),
        options: {
            refetchQueries: [{ query: AllPostsQuery }],
            update: (dataProxy, { data: { updatePost } }) => {
                const query = AllPostsQuery;
                const data = dataProxy.readQuery({ query });

                data.allPost.posts = data.allPost.posts.map(post => post.id !==
updatePost.id ? post : { ...updatePost });

                dataProxy.writeQuery({ query, data });
            }
        }
    })
    )(AllPosts);

    const AddPostWithData = graphql(NewPostMutation, {
    props: (props) => ({
        onAdd: post => props.mutate({
            variables: post,
            optimisticResponse: () => ({ addPost: { ...post, __typename: 'Post', version:
1 } }),
        })
    }),
    options: {
        refetchQueries: [{ query: AllPostsQuery }],
        update: (dataProxy, { data: { addPost } }) => {
        const query = AllPostsQuery;
        const data = dataProxy.readQuery({ query });

        data.allPost.posts.push(addPost);

        dataProxy.writeQuery({ query, data });
        }
    }
})(AddPost);
```

Finally, replace `export default App` with the `ApolloProvider`:

```
const WithProvider = () => (
    <ApolloProvider client={client}>
        <Rehydrated>
            <App />
        </Rehydrated>
    </ApolloProvider>
);

export default WithProvider;
```

Test Your Application

```
yarn start
```

Offline Settings

There are important considerations that you'll need to account for if you want an optimistic UI for an application, where data can be manipulated when the device is in an offline state. Many of these settings are documented in the official Apollo documentation, however, we call out several of them here that you should configure.

First, note that the AWS AppSync client allows you to disable offline capabilities if you simply want to use GraphQL in an always-online scenario. To do this, you pass an additional option when instantiating your client, named `disableOffline`, as follows:

```
const client = new AWSAppSyncClient({
    url: AppSync.graphqlEndpoint,
    region: AppSync.region,
    auth: {
        type: AUTH_TYPE.API_KEY,
        apiKey: AppSync.apiKey,
    },
    disableOffline: true
});
```

- fetchPolicy: This option allows you to specify how a query interacts with the network versus local in-memory caching. AWS AppSync persists this cache to a platform-specific storage medium. If you are using the AWS AppSync client in offline scenarios (`disableOffline:false`), you **MUST** set this value to `cache-and-network`:

  ```
  options: {
    fetchPolicy: 'cache-and-network'
  }
  ```

- optimisticResponse: This option allows you to pass a function or an object to a mutation for updating your UI before the server responds with the result. This is needed in offline scenarios (and for slower networks) to ensure that the UI is updated when the device has no connectivity. Optionally, you can use this if you have set `disableOffline:true`. For example, if you were adding a new object to a list, you might use the following:

  ```
  onAdd: post => props.mutate({
  ```

```
  variables: post,
  optimisticResponse: () => ({ addPost: { __typename: 'Post', ups: 1, downs: 1, content:
 '', url: '', version: 1, ...post } }),
})
```

Normally, you use `optimisticResponse` in conjunction with the `update` option for React Apollo's component, which can trigger during an offline mutation. If you want the UI to update offline for a specific query, you need to specify that query as part of the `readQuery` and `writeQuery` options on the cache, as shown below:

```
options: {
    refetchQueries: [{ query: AllPostsQuery }],
    update: (dataProxy, { data: { addPost } }) => {
        const query = AllPostsQuery;
        const data = dataProxy.readQuery({ query });
        data.allPost.posts.push(addPost);
        dataProxy.writeQuery({ query, data });
      }
   }
```

When this happens, the AWS AppSync persistent store is automatically updated in response to the Apollo cache update. Upon network reconnection, it will synchronize with your GraphQL endpoint. You could also modify more than one query when offline, in which case you could run the above process multiple times in the same `update` block.

Make Your Application Real Time

Edit your schema with the subscription type, as follows:

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

type Mutation {
    addPost(id: ID! author: String! title: String content: String! url: String!): Post!
    updatePost(id: ID! author: String! title: String content: String url: expectedVersion:
 Int!): Post!
    deletePost(id: ID!, expectedVersion: Int): Post
}

type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
}

type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}

type Query {
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID!): Post
}

type Subscription {
    newPost: Post
    @aws_subscribe(mutations:["addPost"])
}
```

Notice that the `@aws_subscribe` specifies which mutations trigger a subscription. You can add more mutations in this array to meet your application needs.

The subscription type `newPost` needs to be passed into an option named `updateQuery` of the React Apollo client to update your UI dynamically when a subcription is received. Ensure that this field name matches the subscription type in the following example code.

In your `App.js` file, edit the `AllPostsWithData` HOC to include `subscribeToNewPost` in the `props` field, as follows:

```
const AllPostsWithData = compose(
    graphql(AllPostsQuery, {
        options: {
            fetchPolicy: 'cache-and-network'
        },
        props: (props) => ({
            posts: props.data.allPost && props.data.allPost.posts,
            // START - NEW PROP :
            subscribeToNewPosts: params => {
                props.data.subscribeToMore({
                    document: NewPostsSubscription,
                    updateQuery: (prev, { subscriptionData: { data : { newPost } } }) => ({
                        ...prev,
                        allPost: { posts: [newPost, ...prev.allPost.posts.filter(post =>
 post.id !== newPost.id)], __typename: 'PaginatedPosts' }
                    })
                });
            },
            // END - NEW PROP
        }
    })
}),
...//more code
```

```
touch ./Queries/NewPostsSubscription.js
```

```
import gql from 'graphql-tag';

export default gql`
subscription NewPostSub {
    newPost {
        __typename
        id
        title
        author
        version
    }
}`;
```

Add the following `import` statement at the top of your `App.js` file:

```
import NewPostsSubscription from './Queries/NewPostsSubscription';
```

Modify the `defaultProps` in the `AllPosts.js` component, as follows:

```
static defaultProps = {
    posts: [],
    onDelete: () => null,
    onEdit: () => null,
    subscribeToNewPosts: () => null,
}
```

Add the following lifecycle method to your `AllPosts` component in `AllPosts.js`:

```
componentWillMount(){
    this.props.subscribeToNewPosts();
}
```

Now try running your app again by typing `yarn start`. Add a new post via the console, with a mutation on `addPost`. You should see real-time data appear in your client application.

Conflict Resolution

When clients make a mutation, either online or offline, they can send a version number with the payload (named `expectedVersion`) for AWS AppSync to check before writing to Amazon DynamoDB. A DynamoDB resolver mapping template can be configured to perform conflict resolution in the cloud, which you can learn about in Resolver Mapping Template Reference for DynamoDB (p. 166). If the service determines it needs to reject the mutation, data is sent to the client and you can optionally run an additional callback to perform client-side conflict resolution.

For example, suppose you had a mutation with DynamoDB set for checking the version, and the client sent `expectedVersion:0`, as in this example:

```
graphql(UpdatePostMutation, {
        props: (props) => ({
        onEdit: (post) => {
            props.mutate({
                variables: { ...post, expectedVersion: 0 },
                optimisticResponse: () => ({ updatePost: { ...post, __typename: 'Post',
 version: post.version + 1 } }),
            })
        }
    }),...more code
```

This would fail the version check because **0** would be lower than any of the current values. You can then define a custom callback conflict resolver. A custom conflict resolver will receive the following variables:

- mutation: GraphQL statement of a mutation
- mutationName: Optional if a name of a mutation is set on a GraphQL statement
- variables: Input parameters of the mutation
- data: Response from AWS AppSync of actual data in DynamoDB
- retries: Number of times a mutation has been retried

For example, you could have the following custom callback conflict resolver:

```
const conflictResolver = ({ mutation, mutationName, variables, data, retries }) => {
    switch (mutationName) {
        case 'UpdatePostMutation':
            return {
                ...variables,
                expectedVersion: data.version,
            };
        default:
            return false;
    }
}
```

In the previous example, you can do a logical check on the `mutationName` and then rerun the mutation with the correct version that AWS AppSync returned.

Note: We recommend doing this only in rare cases. Usually, you should let the AWS AppSync service define conflict resolution, or race conditions can occur. If you don't want to retry, simply return `DISCARD`.

Now, to use this callback, pass it into the AWS AppSync client instantiation:

```
const client = new AWSAppSyncClient({
  url: awsconfig.ENDPOINT,
  region: awsconfig.REGION,
  auth: authInfo,
  conflictResolver,
});
```

Building a JavaScript Client App

This is prerelease documentation for a service in preview release. It is subject to change.

AWS AppSync integrates with the Apollo GraphQL client for building client applications. AWS provides Apollo plugins for offline support, authorization, and subscription handshaking. This tutorial shows how you can use the AWS AppSync SDK with the Apollo client directly in a Node.js application. You can follow a similar process with popular JavaScript frameworks.

Before You Begin

This tutorial expects a GraphQL schema with the following structure:

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

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!
}

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
}

type Subscription {
    newPost: Post
    @aws_subscribe(mutations:["addPost"])
}
```

This schema is from the DynamoDB resolvers tutorial (p. 80), with a subscription added. To follow the complete flow, you can optionally walk through that tutorial first. If you would like to do more customization of GraphQL resolvers, such as those that use DynamoDB, see the Resolver Mapping Template Reference (p. 153).

Get the GraphQL API Endpoint

After you create your GraphQL API, you'll need to get the API endpoint (URL) so you can use it in your client application. There are two ways to get the API endpoint.

In the AWS AppSync console, choose **Home**, and then choose **GraphQL URL** to see the API endpoint.

Alternatively, you can get it by running the following CLI command:

```
aws appsync get-graphql-api --api-id $GRAPHQL_API_ID
```

The following instructions show how you can use AWS_IAM for client authorization. In the console, select **Settings** on the left, and then click **AWS_IAM**.

Create a Client Application

Create a new project and initialize it with npm, accepting the defaults:

```
mkdir appsync && cd appsync
touch index.js aws-exports.js
npm init
```

AWS AppSync supports several authorization types, which you can learn more about in Authorization Use Cases (p. 141). We recommend using short-term credentials from Amazon Cognito Federated Identities or Amazon Cognito user pools. For example purposes, we show how you can use IAM keys. Your `aws-exports` file should look like the following:

```
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
var config = {
    AWS_ACCESS_KEY_ID: '',
    AWS_SECRET_ACCESS_KEY: '',
    HOST: 'URL.YOURREGION.amazonaws.com',
    REGION: 'YOURREGION',
    PATH: '/graphql',
    ENDPOINT: '',
};
config.ENDPOINT = "https://" + config.HOST + config.PATH;
exports.default = config;
```

Edit your `package.json` dependencies file and be sure it includes the following:

```
"dependencies": {
  "apollo-cache-inmemory": "^1.1.0",
  "apollo-client": "^2.0.3",
  "apollo-link": "^1.0.3",
  "apollo-link-http": "^1.2.0",
  "aws-sdk": "^2.141.0",
  "aws-appsync": "^1.0.0",
  "es6-promise": "^4.1.1",
  "graphql": "^0.11.7",
  "graphql-tag": "^2.5.0",
  "isomorphic-fetch": "^2.2.1",
  "ws": "^3.3.1"
}
```

From a command line, run the following:

```
npm install
```

Now add the following to your `index.js` file:

```
"use strict";
/**
* This shows how to use standard Apollo client on Node.js
*/

global.WebSocket = require('ws');
global.window = global.window || {
    setTimeout: setTimeout,
    clearTimeout: clearTimeout,
    WebSocket: global.WebSocket,
    ArrayBuffer: global.ArrayBuffer,
    addEventListener: function () { },
    navigator: { onLine: true }
};
global.localStorage = {
    store: {},
    getItem: function (key) {
        return this.store[key]
    },
    setItem: function (key, value) {
        this.store[key] = value
    },
    removeItem: function (key) {
        delete this.store[key]
    }
};
require('es6-promise').polyfill();
require('isomorphic-fetch');

// Require exports file with endpoint and auth info
const aws_exports = require('./aws-exports').default;

// Require AppSync module
const AUTH_TYPE = require('aws-appsync/lib/link/auth-link').AUTH_TYPE;
const AWSAppSyncClient = require('aws-appsync').default;

const url = aws_exports.ENDPOINT;
const region = aws_exports.REGION;
const type = AUTH_TYPE.AWS_IAM;

// If you want to use API key-based auth
```

```
const apiKey = 'xxxxxxxxx';
// If you want to use a jwtToken from Amazon Cognito identity:
const jwtToken = 'xxxxxxxx';

// If you want to use AWS...
const AWS = require('aws-sdk');
AWS.config.update({
    region: aws_exports.REGION,
    credentials: new AWS.Credentials({
        accessKeyId: aws_exports.AWS_ACCESS_KEY_ID,
        secretAccessKey: aws_exports.AWS_SECRET_ACCESS_KEY
    })
});
const credentials = AWS.config.credentials;

// Import gql helper and craft a GraphQL query
const gql = require('graphql-tag');
const query = gql(`
query AllPosts {
allPost {
    __typename
    id
    title
    content
    author
    version
}
}`);

// Set up a subscription query
const subquery = gql(`
subscription NewPostSub {
newPost {
    __typename
    id
    title
    author
    version
}
}`);

// Set up Apollo client
const client = new AWSAppSyncClient({
    url: url,
    region: region,
    auth: {
        type: type,
        credentials: credentials,
    }
});

client.hydrated().then(function (client) {
    //Now run a query
    client.query({ query: query })
        .then(function logData(data) {
            console.log('results of query: ', data);
        })
        .catch(console.error);

    //Now subscribe to results
    const observable = client.subscribe({ query: subquery });

    const realtimeResults = function realtimeResults(data) {
        console.log('realtime data: ', data);
    };
    observable.subscribe({
        next: realtimeResults,
        complete: console.log,
        error: console.log,
    });
});
```

Notice that in the previous example, if you want to use an API key or Amazon Cognito user pools, you could update the `AUTH_TYPE`:

```
const type = AUTH_TYPE.API_KEY
const type = AUTH_TYPE.AMAZON_COGNITO_USER_POOLS
```

You would need to provide the key or JWT token, as appropriate.

Building an iOS Client App

This is prerelease documentation for a service in preview release. It is subject to change.

AWS AppSync integrates with the Apollo GraphQL client when building client applications. AWS provides plugins for offline support, authorization, and subscription handshaking to make this process easier. You can choose to use the Apollo client directly, or with some client helpers provided in the AWS AppSync SDK when you get started.

Create an API

Before getting started, you will need an API. See Designing a GraphQL API (p. 10) for details, and use the following schema to work with the examples below:

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

    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!
}

    type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
}

type Query {
    posts: [Post]
    post(id: ID!): Post
    searchPosts: [Post]
}

type Subscription {
    newPost: Post
}
```

If you would like to do more customization of GraphQL resolvers, see the Resolver Mapping Template Reference (p. 153).

You will need the endpoint for your client, which you can get from the AWS AppSync console. Under **Home**, look for **GraphQL URL**. Or you can run the following CLI command:

```
aws appsync get-graphql-api --api-id $GRAPHQL_API_ID
```

Download a Client Application

To show usage of AWS AppSync, we first review an iOS application with just a local array of data, and then we add AWS AppSync capabilities to it. Go to the following URL to download a sample application, where we can add, update, and delete posts.

Understanding the iOS Sample App

The iOS sample app has three major files:

1. `PostListViewController` The PostListViewController shows the list of posts available in the app. It uses a simple TableView to list all the posts. You can `Add`, `Update`, or `Delete` posts from this ViewController.
2. `AddPostViewController` The AddPostViewController adds a new post into the list of existing posts. It gives a call to the delegate in PostListViewController to update the list of posts.
3. `UpdatePostViewController` The UpdatePostViewController updates an existing post from the list of posts. It gives a call to the delegate in PostListViewController to update the values of existing posts.

Running the iOS Sample App

1. Open the `PostsApp.xcodeproj` file from the download bundle, which you downloaded in the previous step.
2. Build the project (`COMMAND+B`) and ensure that it completes without error.
3. Run the project (`COMMAND+R`) and try the `Add`, `Update`, and `Delete` (swipe left) operations on the post list.

Set up the Code Generation for GraphQLOperations

To interact with AWS AppSync, your client needs to define GraphQL queries and mutations. This is commonly done in separate files, as follows:

```
mkdir ./GraphQLOperations
touch ./GraphQLOperations/queries.graphql
touch ./GraphQLOperations/mutations.graphql
touch ./GraphQLOperations/subscriptions.graphql
```

Edit and save `queries.graphql`:

```
query post($id:ID!) {
 getPost(id:$id) {
     id
     title
     author
     content
     url
     version
 }
}

query AllPosts {
   allPosts {
       id
       title
       author
       content
       url
       version
   }
}
```

Edit and save `mutations.graphql`:

```
mutation AddPost($id: ID!, $author: String!, $title: String, $url: String,
   $content: String){
  addPost(id:$id, title:$title, author:$author, url:$url, content:$content){
    id
    title
    author
    url
    content
  }
}

mutation UpdatePost($id: ID!, $author: String!, $title: String, $content: String, $url:
 String, $expectedVersion: Int!) {
  updatePost(id: $id, author: $author, title: $title, content: $content, url: $url,
 expectedVersion: $expectedVersion) {
      id
      author
      title
      content
      url
      version
  }
}

mutation DeletePost($id: ID!) {
  deletePost(id:$id){
      id
      title
      author
      url
      content
  }
}
```

Edit and save `subscriptions.graphql`:

```
subscription newPost($author: String) {
 newPost(author: $author) {
   id
   title
   author
   url
   content
   version
 }
}

subscription updatePost {
 updatePost {
    id
    title
    author
    url
    content
    version
 }
}

subscription deletedPost {
 deletePost {
     id
     title
     author
     url
     content
     version
 }
}
```

Run the following commands to install `aws-appsync-codegen` and use the code generator to generate an API for accessing the AWS AppSync backend:

```
npm install -g aws-appsync-codegen

aws-appsync-codegen generate GraphQLOperations/*.graphql --schema GraphQLOperations/
schema.json --output API.swift
```

Add the generated `API.swift` file into your Xcode project. You can make this part of your Xcode build process (p. 67).

Set up Dependency on the AWS AppSync SDK

1. Open a terminal and navigate to the location of the project that you downloaded, and then run the following:

```
pod init
```

This should create a `Podfile` in the root directory of the project. We will use this `Podfile` to declare dependency on the AWS AppSync SDK and other required components.

Open the `Podfile` and add the following lines in the application target:

```
target 'PostsApp' do
  use_frameworks!
  pod 'AWSAppSync' ~> '2.6.7'
end
```

From the terminal, run the following command:

```
pod install --repo-update
```

This should create a file named `PostsApp.xcworkspace`. DO NOT open the `*.xcodeproj` going forward. You can close the `PostsApp.xcodeproj` if it is open.

Open the `PostsApp.xcworkspace` with Xcode. Build the project (`COMMAND+B`) and ensure that it completes without error.

In the app, edit the `Constants.swift` file, and update the GraphQL endpoint and your authentication mechanism.

```
let CognitoIdentityPoolId = "COGNITO_POOL_ID"
let CognitoIdentityRegion: AWSRegionType = .REGION
let AppSyncRegion: AWSRegionType = .REGION
let AppSyncEndpointURL: URL = URL(string: "https://APPSYNCURL/graphql")!
let database_name = "appsync-local-db"
```

Convert the App to Use AWS AppSync for the Backend

Add the `AppSyncClient` as a instance member of the `AppDelegate` class. This enables us to access the same client easily across the app, and update the `didFinishLaunching` method in `AppDelegate.swift` with following code:

```
import AWSAppSync
class AppDelegate {

    var window: UIWindow?
    var appSyncClient: AWSAppSyncClient?

    func application(_ application: UIApplication, didFinishLaunchingWithOptions
 launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
       // Set up  Amazon Cognito credentials
       let credentialsProvider = AWSCognitoCredentialsProvider(regionType:
 CognitoIdentityRegion,
                                                              identityPoolId:
 CognitoIdentityPoolId)
       // You can choose your database location, accessible by SDK
       let databaseURL =
 URL(fileURLWithPath:NSTemporaryDirectory()).appendingPathComponent(database_name)

       do {
           // Initialize the AWS AppSync configuration
           let appSyncConfig = try AWSAppSyncClientConfiguration(url: AppSyncEndpointURL,
                                                                  serviceRegion:
 AppSyncRegion,
                                                                  credentialsProvider:
 credentialsProvider,
                                                                  databaseURL:databaseURL)
           // Initialize the AppSync client
           appSyncClient = try AWSAppSyncClient(appSyncConfig: appSyncConfig)
           // Set id as the cache key for objects
           appSyncClient?.apolloClient?.cacheKeyForObject = { $0["id"] }
       } catch {
           print("Error initializing appsync client. \(error)")
       }
       return true
    }

    // ... other intercept methods
}
```

Update the `AddPostViewController.swift` file with the following code:

```
import Foundation
import UIKit
import AWSAppSync

class AddPostViewController: UIViewController {

    @IBOutlet weak var authorInput: UITextField!
    @IBOutlet weak var titleInput: UITextField!
    @IBOutlet weak var contentInput: UITextField!
    @IBOutlet weak var urlInput: UITextField!
    var appSyncClient: AWSAppSyncClient?

    override func viewDidLoad() {
        super.viewDidLoad()
        let appDelegate = UIApplication.shared.delegate as! AppDelegate
        appSyncClient = appDelegate.appSyncClient!
    }


    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated
    }

    @IBAction func addNewPost(_ sender: Any) {
        // Create a GraphQL mutation
        let uniqueId = UUID().uuidString
        let mutation = AddPostMutation(id: uniqueId,
                                        author: authorInput.text!,
                                        title: titleInput.text,
                                        url: urlInput.text,
                                        content: contentInput.text)

        appSyncClient?.perform(mutation: mutation, optimisticUpdate: { (transaction) in
            do {
                // Update our normalized local store immediately for a responsive UI
                try transaction?.update(query: PostsQuery()) { (data: inout
 PostsQuery.Data) in
                    data.allPosts?.append(PostsQuery.Data.AllPost.init(id: uniqueId, title:
 mutation.title, author: mutation.author, content: mutation.content, version: 0))
                }
            } catch {
                print("Error updating the cache with optimistic response.")
            }
        }) { (result, error) in
            if let error = error as? AWSAppSyncClientError {
                print("Error occurred: \(error.localizedDescription )")
                return
            }
            self.dismiss(animated: true, completion: nil)
        }
        self.dismiss(animated: true, completion: nil)
    }

    @IBAction func onCancel(_ sender: Any) {
        self.dismiss(animated: true, completion: nil)
    }
}
```

Update the `UpdatePostViewController.swift` file with the following code:

```
import Foundation
import UIKit
import AWSAppSync

class UpdatePostViewController: UIViewController {

    var updatePostMutation: UpdatePostMutation?
    @IBOutlet weak var authorInput: UITextField!
    @IBOutlet weak var titleInput: UITextField!
    @IBOutlet weak var contentInput: UITextField!
    @IBOutlet weak var urlInput: UITextField!
    var appSyncClient: AWSAppSyncClient?

    override func viewDidLoad() {
        super.viewDidLoad()
        authorInput.text = updatePostMutation?.author
        titleInput.text = updatePostMutation?.title
        contentInput.text = updatePostMutation?.content
        urlInput.text = updatePostMutation?.url
        let appDelegate = UIApplication.shared.delegate as! AppDelegate
        appSyncClient = appDelegate.appSyncClient!
    }

    @IBAction func updatePost(_ sender: Any) {
        updatePostMutation?.author = authorInput.text!
        updatePostMutation?.title = titleInput.text
        updatePostMutation?.content = contentInput.text
        updatePostMutation?.url = urlInput.text

        appSyncClient?.perform(mutation: updatePostMutation!) { (result, error) in
            if let error = error as? AWSAppSyncClientError {
                print("Error occurred while making request:
 \(error.localizedDescription )")
                return
            }
            if let resultError = result?.errors {
                print("Error saving the item on server: \(resultError)")
                return
            }
            self.dismiss(animated: true, completion: nil)
        }
    }

    @IBAction func onCancel(_ sender: Any) {
        self.dismiss(animated: true, completion: nil)
    }
}
```

Update the `PostListViewController.swift` file with the following code:

```
import UIKit
import AWSAppSync

class PostCell: UITableViewCell {
    @IBOutlet weak var authorLabel: UILabel!
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var contentLabel: UILabel!

    func updateValues(author: String, title:String?, content: String?) {
        authorLabel.text = author
        titleLabel.text = title
        contentLabel.text = content
    }
}
```

```
class PostListViewController: UIViewController, UITableViewDelegate, UITableViewDataSource
 {
    var appSyncClient: AWSAppSyncClient?

    @IBOutlet weak var tableView: UITableView!
    var postList: [PostsQuery.Data.AllPost?]? = [] {
        didSet {
            tableView.reloadData()
        }
    }

    func loadAllPosts() {

        appSyncClient?.fetch(query: PostsQuery(), cachePolicy: .returnCacheDataAndFetch)
 { (result, error) in
            if error != nil {
                print(error?.localizedDescription ?? "")
                return
            }
            self.postList = result?.data?.allPosts
        }
    }

    func loadAllPostsFromCache() {

        appSyncClient?.fetch(query: PostsQuery(), cachePolicy: .returnCacheDataDontFetch)
 { (result, error) in
            if error != nil {
                print(error?.localizedDescription ?? "")
                return
            }
            self.postList = result?.data?.allPosts
        }
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        loadAllPostsFromCache()
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib
        self.automaticallyAdjustsScrollViewInsets = false

        let appDelegate = UIApplication.shared.delegate as! AppDelegate
        appSyncClient = appDelegate.appSyncClient

        loadAllPosts()

        self.tableView.dataSource = self
        self.tableView.delegate = self

        navigationItem.rightBarButtonItem = UIBarButtonItem(title: "Add", style: .plain,
 target: self, action: #selector(addTapped))
    }

    @objc func addTapped() {
        let storyboard = UIStoryboard(name: "Main", bundle: nil)
        let controller = storyboard.instantiateViewController(withIdentifier:
 "NewPostViewController") as! AddPostViewController
        self.present(controller, animated: true, completion: nil)

    }
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return postList?.count ?? 0
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
 UITableViewCell {

        let cell = tableView.dequeueReusableCell(withIdentifier: "PostCell", for:
 indexPath) as! PostCell
        let post = postList![indexPath.row]!
        cell.updateValues(author: post.author, title: post.title, content: post.content)
        return cell
    }

    func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool {
        return true
    }

    func tableView(_ tableView: UITableView, commit editingStyle:
 UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
        if (editingStyle == UITableViewCellEditingStyle.delete) {
            let id = postList![indexPath.row]?.id
            let deletePostMutation = DeletePostMutation(id: id!)
            appSyncClient?.perform(mutation: deletePostMutation) { result, err in
                self.postList?.remove(at: indexPath.row)
            }
            self.tableView.reloadData()
        }
    }

    func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
        let post = postList![indexPath.row]!
        let storyboard = UIStoryboard(name: "Main", bundle: nil)
        let controller = storyboard.instantiateViewController(withIdentifier:
 "UpdatePostViewController") as! UpdatePostViewController
        controller.updatePostMutation = UpdatePostMutation(id: post.id, author:
 post.author, title: post.title, content: post.content, url: post.url, expectedVersion:
 post.version)
        self.present(controller, animated: true, completion: nil)
    }
}
```

Make Your App Real Time

AWS AppSync and GraphQL use the concept of subscriptions to deliver real-time updates of data to the application. We have defined subscriptions on the events of `NewPost`, `UpdatePost`, and `DeletePost`. This means we would get a real-time notification if app data is changed from another device, and we can update our application UI based on the updates.

Add a real-time subscription to receive events on a new post that is added by anyone. In the `PostListViewController.swift` file, add the following function:

```
func startNewPostSubscription() {
  let subscription = NewPostsSubscription()
  do {
  _ = try appSyncClient?.subscribe(subscription: subscription, resultHandler: { (result,
 transaction, error) in
      if let result = result {
          // Store a reference to the new object
          let newPost = result.data!.newPost!
          // Create a new object for the desired query where the new object content should
 reside
          let postToAdd = PostsQuery.Data.Post(id: newPost.id,
                                                    title: newPost.title,
                                                    author: newPost.author,
                                                    content: newPost.content,
                                                    version: newPost.version)
          do {
              // Update the local store with the newly received data
              try transaction?.update(query: PostsQuery()) { (data: inout PostsQuery.Data)
 in
                  data.allPosts?.append(postToAdd)
              }
              self.loadAllPostsFromCache()
          } catch {
              print("Error updating store")
          }
      } else if let error = error {
          print(error.localizedDescription)
      }
  })
  } catch {
    print("Error starting subscription.")
  }
}
```

Next, call the method which we created from the `viewDidLoad` method of `PostListViewController.` This should update the list of posts every time a new post is added from any client.

Integrating into the Build Process

Copy the the `schema.json` file from the download bundle into the application root folder.

The `.graphql` files we created earlier will be used by the `AWS AppSync Codegen` to generate strongly typed API code to perform queries, mutations, and subscriptions. To set up the code generation, we need to add a build step in our Xcode project.

To invoke `AWS AppSync Codegen` as part of the Xcode build process, create a build step that runs before "Compile Sources".

On your application target's **Build Phases** settings tab, click the + icon and choose **New Run Script Phase**. Create a run script, change its name to "Generate AWS Apollo GraphQL API",? and drag it just above "Compile Sources".

Then add the following contents to the script area below the shell:

```
AWS_APOLLO_FRAMEWORK_PATH="$(eval find $FRAMEWORK_SEARCH_PATHS -name "Apollo.framework" -
maxdepth 1)"

if [ -z "$AWS_APOLLO_FRAMEWORK_PATH" ]; then
  echo "error: Couldn't find AWSApollo.framework in FRAMEWORK_SEARCH_PATHS; make sure to
 add the framework to your project."
  exit 1
fi

cd "${SRCROOT}/${TARGET_NAME}"
$AWS_APOLLO_FRAMEWORK_PATH/check-and-run-aws-appsync-codegen.sh generate $(find . -name
 '*.graphql') --schema schema.json --output API.swift
```

The previous script invokes *aws-appsync-codegen* through the *check-and-run-aws-appsync-codegen.sh* wrapper script, which is actually contained in the *AWSApollo.framework* bundle. The main reason for this is to check whether the version of *aws-appsync-codegen* installed on your system is compatible with the framework version installed in your project, and to warn you if it isn't. Without this check, you could end up generating code that is incompatible with the runtime code contained in the framework.

Now build the project using `COMMAND+B` and make sure the `API.swift` file in the project contains generated API code. We will use API code to make GraphQL requests to AWS AppSync.

Complex Objects

Many times you might want to create logical objects that have more complex data, such as images or videos, as part of their structure. For example, you might create a "Person" type with a profile picture or a "Post" type that has an associated image. You can use AWS AppSync to model these as GraphQL types. If any of your mutations have a variable with `bucket`, `key`, `region`, `mimeType`, and `localUri` fields, the SDK will upload the file to Amazon S3 for you.

Edit your schema as follows to add the `S3Object` and `S3ObjectInput` types:

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

type Mutation {
    addPost(id: ID! author: String! title: String content: String url: String file:
 S3ObjectInput): Post!
    updatePost(id: ID! author: String! title: String content: String url: String ups: Int!
 downs: Int! expectedVersion: Int!): Post!
    deletePost(id: ID!): Post!
}

type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    file: S3Object
    version: Int!
}

type S3Object {
    bucket: String!
    key: String!
    region: String!
}

input S3ObjectInput {
    bucket: String!
    key: String!
    region: String!
    localUri: String
    mimeType: String
}

type Query {
    allPost: [Post]
    getPost(id: ID!): Post
}

type Subscription {
    newPost: Post
    @aws_subscribe(mutations:["addPost"])
}
```

The AWS AppSync SDK does not take a direct dependency to the AWS SDK for iOS for S3, but takes in `AWSS3TransferUtility` and `AWSS3PresignedURLClient` clients as part of `AWSAppSyncClientConfiguration`. The code generator used above for generating the API will generate the S3 wrappers required to use the previous clients in the client code. To generate the wrappers, pass the `--add-s3-wrapper` flag while running the code generator tool. You will also need to take a dependency on `AWSS3` SDK. You can do that by updating your `Podfile` to the following:

```
target 'PostsApp' do
  use_frameworks!
  pod 'AWSAppSync' ~> '2.6.7'
  pod 'AWSS3' ~> '2.6.7'
end
```

Then run `pod install` to fetch the new dependency.

Update the `AWSAppSyncClientConfiguration` object to provide the `AWSS3TransferUtility` client for managing the uploads and downloads.

```
let appSyncConfig = try AWSAppSyncClientConfiguration(url: AppSyncEndpointURL,
                                                      serviceRegion: AppSyncRegion,
                                                      credentialsProvider:
 credentialsProvider,
                                                      databaseURL:databaseURL,
                                                      s3ObjectManager:
 AWSS3TransferUtility.default())
```

The mutation operation does not require any specific changes in method signature, but requires only an `S3ObjectInput` with `bucket`, `key`, `region`, `localUri`, and `mimeType`. Now when you do a mutation, it automatically uploads the specified file to S3 using the `AWSS3TransferUtility` client internally.

Conflict Resolution

When clients make a mutation, either online or offline, they can send a version number with the payload (named `expectedVersion`) for AWS AppSync to check before writing to Amazon DynamoDB. A DynamoDB resolver mapping template can be configured to perform conflict resolution in the cloud. See Resolver Mapping Template Reference for DynamoDB (p. 166). If the service determines it needs to reject the mutation, data is sent to the client and you can optionally run an additional callback to perform client-side conflict resolution.

For example, suppose you had a mutation with DynamoDB set for checking the version, and the client sent `expectedVersion:0`, as in this example:

```
@IBAction func updatePost(_ sender: Any) {
    let updatePostMutation = UpdatePostMutation(id: "1",
                                                author: "Mr. Abc",
                                                content: "UpdatedContent",
                                                expectedVersion: 0)


    appSyncClient?.perform(mutation: updatePostMutation) { (result, error) in
        if let error = error as? AWSAppSyncClientError {
            print("Error occurred while making request: \(error.localizedDescription )")
            return
        }
        if let resultError = result?.errors {
            print("Error saving the item on server: \(resultError)")
            return
        }
        self.dismiss(animated: true, completion: nil)
    }
}
```

This would fail the version check because **0** would be lower than any of the current values. You can then define a custom callback conflict resolver. A custom conflict resolver can be passed inline to resolve conflicts:

```
appSyncClient?.perform(mutation: updatePostMutation, conflictResolutionBlock:
 { (serverState, taskCompletionSource, result) in
          // conflict resolution block gets a callback here
          let snapshot = UpdatePostMutation.Data.UpdatePost(snapshot: serverState!)
          print("Server version is: \(snapshot.version)")
          let updateMutation = UpdatePostMutation(id: "1", author: "Mr. Abc", content:
 "UpdatedContent", expectedVersion: snapshot.version)
          // this would retry the specified `updateMutation` before processing any other
 queued mutations.
          taskCompletionSource?.set(result: updateMutation)
      }, resultHandler: { (result, error) in
          if let error = error as? AWSAppSyncClientError {
              print("Error occurred while making request: \(error.localizedDescription )")
              return
          }
          if let resultError = result?.errors {
              print("Error saving the item on server: \(resultError)")
              return
          }
          self.dismiss(animated: true, completion: nil)
      })
```

Building an Android Client App

This is prerelease documentation for a service in preview release. It is subject to change.

Create an API

Before getting started, you will need an API. See Designing a GraphQL API for details, and use the following schema to work with the examples below:

```
type Mutation {
    deletePost(id: ID!): Post!
    putPost(
        id: ID!,
        author: String!,
        title: String,
        content: String,
        url: String,
        ups: Int,
        downs: Int,
        version: Int!
    ): Post
}
```

```
type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
}

type Query {
    getPost(id: ID!): Post
    allPost(count: Int, nextToken: String): [Post]
}

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

If you want to do more customization of GraphQL resolvers, see the Resolver Mapping Template Reference (p. 153).

Download a Client Application

This tutorial will use the AWS AppSyncPosts schema starter kit

If you wish to see the entire app without following the steps, here is the whole sample

Gradle Setup

Project's build.gradle

In the project's `build.gradle` file, add the following dependency in the build script:

```
classpath 'com.amazonaws:aws-android-sdk-appsync-gradle-plugin:2.6.15'
```

Sample Project's build.gradle

```
// Top-level build file where you can add configuration options common to all sub-projects/
modules.
buildscript {
    // ..other code..
    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.1'
        classpath 'com.amazonaws:aws-android-sdk-appsync-gradle-plugin:2.6.15'
        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}
```

App's build.gradle

In the app's `build.gradle` file, add the following plugin:

```
apply plugin: 'com.amazonaws.appsync'
```

Add the following dependency:

```
compile 'com.amazonaws:aws-android-sdk-appsync:2.6.15'
```

Sample App's build.gradle

```
apply plugin: 'com.android.application'
apply plugin: 'com.amazonaws.appsync'
android {
    // Typical items
}
dependencies {
    // Typical dependencies
    compile 'com.amazonaws:aws-android-sdk-appsync:2.6.15'
}
```

App's AndroidManifest.xml

Add the permissions to access the network state to determine if the device is offline.

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Code Generation for GraphQL Operations

To interact with AWS AppSync, your client needs to define GraphQL queries and mutations.

Create a file named `./app/src/main/graphql/com/amazonaws/demo/posts/posts.graphql`:

```
query GetPost($id:ID!) {
 getPost(id:$id) {
     id
     title
     author
     content
     url
     version
 }
}

query AllPosts {
   allPost {
       id
       title
       author
       content
       url
       version
       ups
       downs
   }
}

mutation AddPost($id: ID!, $author: String!, $title: String, $content: String, $url:
 String, $ups: Int!, $downs: Int!, $expectedVersion: Int!) {
  putPost(id: $id, author: $author, title: $title, content: $content, url: $url, ups: $ups,
 downs: $downs, version: $expectedVersion) {
    id
    title
    author
    url
    content
  }
}

mutation UpdatePost($id: ID!, $author: String!, $title: String, $content: String, $url:
 String, $ups: Int!, $downs: Int!, $expectedVersion: Int!) {
  putPost(id: $id, author: $author, title: $title, content: $content, url: $url, ups: $ups,
 downs: $downs, version: $expectedVersion) {
      id
      author
      title
      content
      url
      version
  }
}

mutation DeletePost($id: ID!) {
  deletePost(id:$id){
      id
      title
      author
      url
      content
  }
}
```

Next, fetch the `schema.json` file from the AWS AppSync console and place it alongside the `posts.graphql` file. `./app/src/main/graphql/com/amazonaws/demo/posts/schema.json`

Now build the project. The generated source files will be available to use within the app. They will not show up in your source directory, but are added in the build path.

Call the Service

In this section, we create a client and call the service.

Set up Constants.java

You will need the endpoint for your client, which you can get from the AWS AppSync console. Under **Home**, look for **API URL**. Or you can run the following CLI command:

```
aws appsync get-graphql-api --api-id $GRAPHQL_API_ID
```

Edit `Constants.java` with your information. Not all of the constants will be used, depending on the authorization path chosen in the next steps. For example, if you're using API key-based authorization, the identity pool ID and region can be ignored.

```
public static final String APPSYNC_API_URL = "https://API-URL/graphql"; // TODO: Update the
 endpoint URL as specified on AppSync console

// API Key Authorization
public static final String APPSYNC_API_KEY = "API-KEY"; // TODO: Copy the API Key specified
 on the AppSync Console

// IAM based Authorization (Cognito Identity)
public static final String COGNITO_IDENTITY = "POOL-ID"; // TODO: Update the Cognito
 Identity Pool ID
public static final Regions COGNITO_REGION = Regions.US_WEST_2; // TODO: Update the region
 to match the Cognito Identity Pool region
```

```
// Cognito User Pools Authorization
public static final String USER_POOLS_POOL_ID = "";
public static final String USER_POOLS_CLIENT_ID = "";
public static final String USER_POOLS_CLIENT_SECRET = "";
public static final Regions USER_POOLS_REGION = Regions.US_WEST_2; // TODO: Update the
 region to match the Cognito User Pools region
```

Create the client

When making calls to AWS AppSync, there are 3 ways to authenticate those calls. The API Key Authorization is the most simple to on-board with, but it is recommended to use either IAM or Amazon Cognito User Pools after on-boarding with an API key.

API Key Authorization

For authorization using the API key, update the `ClientFactory.getInstance(Context)` method with the following code.

```
public class ClientFactory {

    // ...other code...
    private static volatile AWSAppSyncClient client;

    public static AWSAppSyncClient getInstance(Context context) {
      if (client == null) {
        client = AWSAppSyncClient.builder()
                .context(context)
                .apiKey(new BasicAPIKeyAuthProvider(Constants.APPSYNC_API_KEY)) // API Key
 based authorization
                .region(Constants.APPSYNC_REGION)
                .serverUrl(Constants.APPSYNC_API_URL)
                .build();
      }
      return client;
    }
}
```

IAM-Based Authorization (Amazon Cognito Identity)

For IAM-based authorization, update the `ClientFactory.getInstance(Context)` method with the following code.

```
public class ClientFactory {

    // ...other code...
    private static volatile AWSAppSyncClient client;

    public static AWSAppSyncClient getInstance(Context context) {
      if (client == null) {
        client = AWSAppSyncClient.builder()
                .context(context)
                .credentialsProvider(getCredentialsProvider(context)) // For use with IAM/
Cognito authorization
                .region(Constants.APPSYNC_REGION)
                .serverUrl(Constants.APPSYNC_API_URL)
                .build();
      }
      return client;
    }
    private static final AWSCredentialsProvider getCredentialsProvider(final Context
 context) {
        final CognitoCachingCredentialsProvider credentialsProvider = new
 CognitoCachingCredentialsProvider(
                context,
                Constants.COGNITO_IDENTITY,
                Constants.COGNITO_REGION
        );
        return credentialsProvider;
    }
}
```

Cognito User Pools Authorization

For Cognito User Pools Authorization, update the `ClientFactory.getInstance(Context)` method with the following code.

```
import com.amazonaws.mobileconnectors.cognitoidentityprovider.CognitoUserPool;

public class ClientFactory {

    // ...other code...
    private static volatile AWSAppSyncClient client;

    public static AWSAppSyncClient getInstance(Context context) {
      if (client == null) {
        client = AWSAppSyncClient.builder()
                .context(context)
                .cognitoUserPoolsAuthProvider(getUserPools(context)) // For use with User
 Pools authorization
                .region(Constants.APPSYNC_REGION)
                .serverUrl(Constants.APPSYNC_API_URL)
                .build();
      }
      return client;
    }

    private static CognitoUserPoolsAuthProvider getUserPools(final Context context) {
        final CognitoUserPoolsAuthProvider provider = new
 BasicCognitoUserPoolsAuthProvider(new CognitoUserPool(
                context,
                Constants.USER_POOLS_POOL_ID,
                Constants.USER_POOLS_CLIENT_ID,
                Constants.USER_POOLS_CLIENT_SECRET,
                Constants.USER_POOLS_REGION));
        return provider;
    }
}
```

Query the Posts

Add the `PostsActivity.queryData()` method and create a callback to receive the data, when available.

```
public class PostsActivity extends AppCompatActivity {

    // ...other code...

    public void queryData() {
        if (mAWSAppSyncClient == null) {
            mAWSAppSyncClient = ClientFactory.getInstance(this);
        }
        mAWSAppSyncClient.query(AllPostsQuery.builder().build())
                .responseFetcher(AppSyncResponseFetchers.CACHE_AND_NETWORK)
                .enqueue(postsCallback);
    }

    private GraphQLCall.Callback<AllPostsQuery.Data> postsCallback = new
 GraphQLCall.Callback <AllPostsQuery.Data>() {
        @Override
        public void onResponse(@Nonnull final Response<AllPostsQuery.Data> response) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    PostsActivity.this.mAdapter.setPosts(response.data().allPost());
                    PostsActivity.this.mSwipeRefreshLayout.setRefreshing(false);
                    PostsActivity.this.mAdapter.notifyDataSetChanged();
                }
            });
        }

        @Override
        public void onFailure(@Nonnull ApolloException e) {
            Log.e(TAG, "Failed to perform AllPostsQuery", e);
            PostsActivity.this.mSwipeRefreshLayout.setRefreshing(false);
        }
    };
}
```

Mutate the Posts (Add a Post)

Add the `AddPostActivity.save()` method and create a callback to receive confirmation.

```
public class AddPostActivity extends AppCompatActivity {

    // ...other code...

    private void save() {
        final String title = ((EditText)
 findViewById(R.id.updateTitle)).getText().toString();
        final String author = ((EditText)
 findViewById(R.id.updateAuthor)).getText().toString();
        final String url = ((EditText) findViewById(R.id.updateUrl)).getText().toString();
        final String content = ((EditText)
 findViewById(R.id.updateContent)).getText().toString();

        AddPostMutation addPostMutation = AddPostMutation.builder()
                .id(UUID.randomUUID().toString())
                .title(title)
                .author(author)
                .url(url)
                .content(content)
                .ups(0)
                .downs(0)
                .expectedVersion(1)
                .build();
        ClientFactory.getInstance(this).mutate(addPostMutation).enqueue(postsCallback);
    }

    private GraphQLCall.Callback<AddPostMutation.Data> postsCallback = new
 GraphQLCall.Callback<AddPostMutation.Data>() {
        @Override
        public void onResponse(@Nonnull final Response<AddPostMutation.Data> response) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Toast.makeText(AddPostActivity.this, "Added post",
 Toast.LENGTH_SHORT).show();
                    AddPostActivity.this.finish();
                }
            });
        }

        @Override
        public void onFailure(@Nonnull final ApolloException e) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Log.e("", "Failed to perform AddPostMutation", e);
                    Toast.makeText(AddPostActivity.this, "Failed to add post",
 Toast.LENGTH_SHORT).show();
                    AddPostActivity.this.finish();
                }
            });
        }
    };
}
```

Mutate the Posts (Update a Post)

Add the `UpdatePostActivity.save()` method and create a callback to receive confirmation.

```
public class UpdatePostActivity extends AppCompatActivity {

    // ...other code...

    private void save() {
        final String title = ((EditText)
 findViewById(R.id.updateTitle)).getText().toString();
        final String author = ((EditText)
 findViewById(R.id.updateAuthor)).getText().toString();
        final String url = ((EditText) findViewById(R.id.updateUrl)).getText().toString();
        final String content = ((EditText)
 findViewById(R.id.updateContent)).getText().toString();

        UpdatePostMutation updatePostMutation = UpdatePostMutation.builder()
                .id(sPost.id())
                .title(title)
                .author(author)
                .url(url)
                .content(content)
                .ups(sPost.ups())
                .downs(sPost.downs())
                .expectedVersion(sPost.version() + 1)
                .build();

        // Make mutation call
        ClientFactory.getInstance(this).mutate(updatePostMutation).enqueue(postsCallback);
    }

    private GraphQLCall.Callback<UpdatePostMutation.Data> postsCallback = new
 GraphQLCall.Callback<UpdatePostMutation.Data>() {
        @Override
        public void onResponse(@Nonnull Response<UpdatePostMutation.Data> response) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Toast.makeText(UpdatePostActivity.this, "Updated post",
 Toast.LENGTH_SHORT).show();
                    UpdatePostActivity.this.finish();
                }
            });
        }

        @Override
        public void onFailure(@Nonnull final ApolloException e) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Log.e("", "Failed to perform UpdatePostMutation", e);
                    Toast.makeText(UpdatePostActivity.this, "Failed to update post",
 Toast.LENGTH_SHORT).show();
                    UpdatePostActivity.this.finish();
                }
            });
        }
    };
}
```

Optimistic Updates

This section makes changes to the `UpdatePostActivity.save()` method that was created in the previous step.

For optimistic updates, create the data expected to be returned after the mutation. The optimistic updates are written to the persistent SQL store.

```
UpdatePostMutation.Data expected = new UpdatePostMutation.Data(new
 UpdatePostMutation.PutPost("Post", sPost.id(),author,title,content,url,sPost.version() +
 1));
```

Replace the mutation call with the following, which takes in the expected data:

```
// Make mutation call
ClientFactory.createInstance(this).mutate(updatePostMutation,
 expected).enqueue(postsCallback);
//                                                              ^^^^^^^^
```

Offline Mutations

Offline mutations currently work out of the box and are available in memory, as well as through app restarts.

The callback for `onResponse` is received when the network is available, and the request goes through if the app was not killed.

For mutations which are performed after an app restart, the `PersistentMutationsCallback` object will be called. The
<problematic>``</problematic>
PersistentMutationsCallback will have information about the mutation type and identifier. It can be specified while initializing the client.

```
AWSAppSyncClient client = AWSAppSyncClient.builder()
    .context(context)
    .apiKey(new BasicAPIKeyAuthProvider(Constants.APPSYNC_API_KEY)) // API Key based auth
    .region(Constants.APPSYNC_REGION)
    .serverUrl(Constants.APPSYNC_API_URL)
    .persistentMutationsCallback(new PersistentMutationsCallback() {
```

```
                    @Override
                    public void onResponse(PersistentMutationsResponse response) {
                        if (response.getMutationClassName().equals("AddPostMutation")) {
                            // perform action here add post mutation
                        }
                    }

                    @Override
                    public void onFailure(PersistentMutationsError error) {
                        // handle error feedback here
                    }
                })
    .build();
```

Data Sources and Resolvers

This is prerelease documentation for a service in preview release. It is subject to change.

AWS AppSync supports the automatic provisioning of DynamoDB tables from a GraphQL schema as described in (Optional) Provision from Schema (p. 25) and Launch a Sample Schema (p. 2). You can use a GraphQL API with your existing AWS resources or build data sources and resolvers. This section takes you through this process in a series of tutorials.

Topics

Tutorial: DynamoDB Resolvers

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 provision AWS resources using the following AWS CloudFormation template:

```
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
```

You can launch this AWS CloudFormation stack in the US West 2 (Oregon) region in your AWS account by clicking this button:

Launch Stack

This will create:

- 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.

To see more details about the stack and the created resources, run the following CLI command:

```
aws cloudformation describe-stacks \
    --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

To delete the resources later, you can run:

```
aws cloudformation delete-stack \
    --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

Creating Your GraphQL API

To create the GraphQL API in AWS AppSync:

- Open the AWS AppSync console and choose the **Create API** button.
- Set the name of the API to `AWSAppSyncTutorial`.
- Select **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 against it for the rest of this tutorial.

Defining a Basic "Post" API

Now that you set up an AWS AppSync GraphQL API, you can set up a basic schema that allows the basic creation, retrieval, and deletion of post data.

In the AWS AppSync console, choose the **Schema** tab, replace the contents of the **Schema** pane with the following code, and then choose the **Save**.

```
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!
}
```

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-Post`DynamoDB table.

First, AWS AppSync needs to be aware of your tables. You do this by setting up a data source in AWS AppSync:

- Go to the **Data source** tab.
- Choose **New** to create a new data source.
- Enter `PostDynamoDBTable` as the name of the data source.
- Choose **Amazon DynamoDB table** as the data source type.
- Choose **US-WEST-2** for the region.
- Choose the **AppSyncTutorial-Post**DynamoDB table from the list of tables.
- Choose **Existing role** in the **Create or use an existing role** section.
- Select the `arn:aws:iam::123456789012:role/AppSyncTutorialAmazonDynamoDBRole` role from the list of available roles.
- 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:

- Go to the **Schema** tab.

- Find the `addPost` field on the `Mutation` type in the **Data types** pane on the right.
- Click on its `Attach` button.
- Select `PostDynamoDBTable` in the **Data source name** drop-down menu.
- Paste the following into the **Configure the request mapping template** section:

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : { "S" : "${context.arguments.id}" }
    },
    "attributeValues" : {
        "author": { "S" : "${context.arguments.author}" },
        "title": { "S" : "${context.arguments.title}" },
        "content": { "S" : "${context.arguments.content}" },
        "url": { "S" : "${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 we'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 `$utils.autoId()` which you could have also used in the form of `"id" : { "S" : "${context.arguments.id}" }`. Then you could simply leave the `id: ID!` out of the schema definition of `addPost()` and it would be inserted automatically. We won't use this technique for this tutorial, but you should consider it as a good practice when writing to DynamoDB tables.

  See the Resolver Mapping Template Overview (p. 153) reference documentation for more information about mapping templates, see the GetItem (p. 167) reference documentation for more information about GetItem request mapping, and see the Type System (Request Mapping) (p. 180) reference documentation for more info about types.
- Paste the following into the **Configure the response mapping template** section:

```
$utils.toJson($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.
- 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.

- Go to the **Queries** tab

- Paste the following mutation into the **Queries** pane

```
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
  }
}
```

- Then choose the **Execute query** button (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 something like this:

```
{
  "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 we're able to add data to the `AppSyncTutorial-Post`DynamoDB table, we need to set up the `getPost` query so it can retrieve that data from the `AppSyncTutorial-Post` table. To do this, we set up another resolver.

- Go to the "Schema" tab.
- Find the `getPost` field on the `Query` type in the "Data types" pane on the right.
- Click on its `Attach` button.
- Select `PostDynamoDBTable` in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "${context.arguments.id}" }
    }
}
```

- Paste the following into the "Configure the response mapping template" section:

```
$utils.toJson($context.result)
```

- Click the `Save` button.

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. We can now run a query to retrieve the post we created earlier.

- Go to the "Queries" tab
- Paste the following mutation into the "Queries" pane.

```
query getPost {
  getPost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Then hit the "Execute query" button (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 something like this:

```
{
  "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 an `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.

Create an updatePost mutation (DynamoDB UpdateItem)

So far we can create and retrieve `Post` objects in DynamoDB. Next, we'll set up a new mutation to allow us to update object. We'll do this using the UpdateItem DynamoDB operation.

- Go to the "Schema" tab
- Modify the `Mutation` type in the "Schema" pane to add a new `updatePost` mutation:

```
type Mutation {
    updatePost(
        id: ID!,
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post
    addPost(
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}
```

- Click the `Save` button
- Find the newly created `updatePost` field on the `Mutation` type in the "Data types" pane on the right.
- Click on its `Attach` button.
- Select `PostDynamoDBTable` in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : { "S" : "${context.arguments.id}" }
    },
    "update" : {
        "expression" : "SET author = :author, title = :title, content = :content, #url
 = :url ADD version :one",
        "expressionNames": {
            "#url" : "url"
        },
        "expressionValues": {
            ":author" : { "S": "${context.arguments.author}" },
            ":title" : { "S": "${context.arguments.title}" },
            ":content" : { "S": "${context.arguments.content}" },
            ":url" : { "S": "${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, we'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 we can use name placeholders and define them in the `expressionNames` field.

See the UpdateItem (p. 170) reference documentation for more info about UpdateItem request mapping, and the DynamoDB UpdateExpressions documentation for more information on how to write update expressions.

- Paste the following into the "Configure the response mapping template" section:

```
$utils.toJson($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. We can now run a mutation to update the item we wrote earlier.

- Go to the "Queries" tab
- Paste the following mutation into the "Queries" pane. You'll also need to update the `id` argument to have the value we 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
  }
}
```

- Then hit the "Execute query" button (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 something like this:

```
{
  "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
    }
  }
}
```

Note that the `ups` and `downs` fields were not modified. This is because our request mapping template did not ask AWS AppSync and DynamoDB to do anything with those fields. Also note that the `version` field was incremented by 1. This is because we asked AWS AppSync and DynamoDB to add 1 to the `version` field.

Modifying the "updatePost" resolver (DynamoDB UpdateItem)

This is a good start to our `updatePost` mutation, but it has two main problems:

- If I want to update just a single field, then I have to update all the fields.
- If two people are modifying the object, then we potentially lose information.

To address these issues, we'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.

- Go to the "Schema" tab.
- Modify the `updatePost` field in the `Mutation` type in the "Schema" pane 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!
}
```

- Click the `Save` button
- Find the `updatePost` field on the `Mutation` type in the "Data types" pane on the right.
- Click on the `PostDynamoDBTable` link to open the existing resolver.
- Modify the request mapping template in the "Configure the request mapping template" section:

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : { "S" : "${context.arguments.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", ":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 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" : { "N" : ${context.arguments.expectedVersion} }
        }
    }
}
```

- Click the `Save` button.

This template is one of the more complex examples, but demonstrates the power and flexibility of mapping templates. What it is doing is looping through all the arguments, skipping over `id` and `expectedVersion`. If the argument is set to something, then it will ask AWS AppSync and DynamoDB to update that attribute on the object in DynamoDB. If the attribute is set to null, then it will ask AWS AppSync and DynamoDB to remove that attribute from the post object. If an argument wasn't specified, then it will leave it alone. It also increments the `version` field.

There is also a new `condition` section. A condition expression let you tell AWS AppSync and DynamoDB whether the request should succeed or not based on the state of the object already in DynamoDB before the operation is performed. In this case, we only want the UpdateItem request to succeed if the `version` field of the item currently in DynamoDB exactly matches the `expectedVersion` argument.

See the Condition Expressions (p. 188) reference documentation for more information about condition expressions.

Call the API to update a Post

Lets try updating our Post object with the new resolver:

- Go to the "Queries" tab
- Paste the following mutation into the "Queries" pane. You'll also need to update the `id` argument to have the value we noted down earlier.

```
mutation updatePost {
  updatePost(
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 2
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Then hit the "Execute query" button (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 something like this:

```
{
  "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
    }
  }
}
```

Note that in this request, we only asked AWS AppSync and DynamoDB to update the `title` and `content` field. It left all the other fields alone (other than incrementing the `version` field). We 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 will 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 we 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 we 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 AWS AppSync's 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 our case, we can see the `version` field of the object in DynamoDB is set to `3`, so we could just update the `expectedVersion` argument to `3` and the request would succeed again.

See the Condition Expressions (p. 188) mapping template reference documentation for more information about handling condition check failures.

Create upvotePost and downvotePost mutations (DynamoDB UpdateItem)

Our `Post` type has `ups` and `downs` fields to let us record upvotes and downvotes, but so far our API doesn't let us do anything with them. Lets add some mutations to let us upvote and downvote our posts.

- Go to the "Schema" tab
- Modify the `Mutation` type in the "Schema" pane to add new `upvotePost` and `downvotePost` mutations:

```
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!
}
```

- Click the `Save` button
- Find the newly created `upvotePost` field on the `Mutation` type in the "Data types" pane on the right.
- Click on its `Attach` button.
- Select `PostDynamoDBTable` in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : { "S" : "${context.arguments.id}" }
    },
    "update" : {
        "expression" : "ADD ups :plusOne, version :plusOne",
        "expressionValues" : {
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- Paste the following into the "Configure the response mapping template" section:

```
$utils.toJson($context.result)
```

- Click the `Save` button.
- Find the newly created `downvotePost` field on the `Mutation` type in the "Data types" pane on the right.
- Click on its `Attach` button.
- Select `PostDynamoDBTable` in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : { "S" : "${context.arguments.id}" }
    },
    "update" : {
        "expression" : "ADD downs :plusOne, version :plusOne",
        "expressionValues" : {
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- Paste the following into the "Configure the response mapping template" section:

```
$utils.toJson($context.result)
```

- Click the `Save` button.

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. We can now run mutations to upvote or downvote the post we created earlier.

- Go to the "Queries" tab
- Paste the following mutation into the "Queries" pane. You'll also need to update the `id` argument to have the value we noted down earlier.

```
mutation votePost {
  upvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Then hit the "Execute query" button (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 something like this:

```
{
  "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
    }
  }
}
```

- Click the "Execute query" button 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:

```
mutation votePost {
  downvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Then hit the "Execute query" button (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 DeletePost)

The next mutation we want to set up is to delete a post. We'll do this using the DeleteItem DynamoDB operation.

- Go to the "Schema" tab
- Modify the `Mutation` type in the "Schema" pane to add a new `deletePost` mutation:

```
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!
}
```

  Note that this time we made the `expectedVersion` field optional. The reason for this will be explained when we add the request mapping template.
- Click the `Save` button
- Find the newly created `delete` field on the `Mutation` type in the "Data types" pane on the right.
- Click on its `Attach` button.
- Select `PostDynamoDBTable` in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```
{
    "version" : "2017-02-28",
    "operation" : "DeleteItem",
    "key": {
        "id": { "S" : "${context.arguments.id}"}
    }
    #if( $context.arguments.containsKey("expectedVersion") )
        ,"condition" : {
            "expression"       : "attribute_not_exists(id) OR version
 = :expectedVersion",
            "expressionValues" : {
                ":expectedVersion" : { "N" : ${context.arguments.expectedVersion} }
            }
        }
    #end
}
```

Note: The `expectedVersion` argument is an optional argument. If the caller set an `expectedVersion` argument in the request, then the template will add in a condition that will only allow the DeleteItem request to succeed if the item is already deleted, or the `version` attribute of the post in DynamoDB exactly matches the `expectedVersion`. If left out, no condition expression is specified on the DeleteItem request, and it will succeed regardless of the value of `version` or if the item exists in DynamoDB or not.

- Paste the following into the "Configure the response mapping template" section:

```
$utils.toJson($context.result)
```

Note: Even though we're deleting an item, we can return the item that was deleted, if it was not already deleted.

- Click the `Save` button.

See the DeleteItem (p. 173) reference documentation for more info about DeleteItem request mapping.

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. We can now run a mutation to delete something in the table.

- Go to the "Queries" tab
- Paste the following mutation into the "Queries" pane. You'll also need to update the `id` argument to have the value we noted down earlier.

```
mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Then hit the "Execute query" button (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 something like this:

```
{
  "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.

- Try hitting the "Execute query" button again.
- The call still succeeds, but no value is returned.

```
{
  "data": {
    "deletePost": null
  }
}
```

Now lets try deleting a post, but this time specifying an `expectedValue`. First though, we'll need to create a new post because we've just deleted the one we've been working with so far.

- Paste the following mutation into the "Queries" pane

```
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
  }
}
```

- Then hit the "Execute query" button (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, as we'll need it in just a moment. It should look something like this:

```
{
  "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 lets try and delete that post, but we'll put in the wrong value for `expectedVersion`

- Paste the following mutation into the "Queries" pane. You'll also need to update the `id` argument to have the value we noted down earlier.

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Then hit the "Execute query" button (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
  }
}
```

- Then hit the "Execute query" button (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
    }
  }
}
```

- Try hitting the "Execute query" button 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 our API is only useful if we know the `id`'s of each post we want to look at. Lets add a new resolver that will return all the posts in the table.

- Go to the "Schema" tab
- Modify the `Query` type in the "Schema" pane to add a new `allPost` query:

```
type Query {
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

- Add a new `PaginationPosts` type in the "Schema pane:

```
type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}
```

- Click the `Save` button
- Find the newly created `allPost` field on the `Query` type in the "Data types" pane on the right.
- Click on its `Attach` button.
- Select `PostDynamoDBTable` in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
    #if( ${context.arguments.count} )
        ,"limit": ${context.arguments.count}
    #end
    #if( ${context.arguments.nextToken} )
        ,"nextToken": "${context.arguments.nextToken}"
    #end
}
```

  Note that 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 (we'll show where the value for `nextToken` comes from later).
- Paste the following into the "Configure the response mapping template" section:

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": "${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`.
- Click the `Save` button.

See the Scan (p. 178) reference documentation for more info about Scan request mapping.

Call the API to scan all Posts

Now the resolver has been set up, AWS AppSync knows how to translate an incoming `allPost` mutation to a DynamoDB Scan operation. We can now scan the table to retrieve all the posts.

Before we can try it out though, we need to populate the table with some data, because we've deleted everything we've worked with so far.

- Go to the "Queries" tab
- Paste the following mutation into the "Queries" pane

```
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 }
}
```

- Then hit the "Execute query" button (the orange play button).

Now, lets scan the table, returning 5 results at a time.

- Paste the following query in the "Queries" pane

```
query allPost {
  allPost(count: 5) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Then hit the "Execute query" button (the orange play button).
- The first 5 posts should appear in the results pane to the right of the query pane. It should look something like this:

```
{
  "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":
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI3Z0lnRkJ
    }
  }
}
```

We can see that we got 5 results and also a `nextToken` that we 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:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI3Z0lnRll
  ) {
    posts {
      id
      author
    }
    nextToken
  }
}
```

- Then hit the "Execute query" button (the orange play button).
- The remaining 4 posts should appear in the results pane to the right of the query pane. There is no `nextToken` in this set of results as we have paged through all 9 posts, with none remaining. It should look something like this:

```
{
  "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, we can also query DynamoDB to retrieve posts created by a specific author. The DynamoDB table we created earlier already has a GlobalSecondaryIndex called `author-index` we can use with a DynamoDB Query operation to retrieve all posts created by a specific author.

- Go to the "Schema" tab
- Modify the `Query` type in the "Schema" pane to add a new `allPostsByAuthor` query:

```
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 we used with the `allPost` query.
- Click the `Save` button
- Find the newly created `allPostsByAuthor` field on the `Query` type in the "Data types" pane on the right.
- Click on its `Attach` button.
- Select `PostDynamoDBTable` in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "index" : "author-index",
    "query" : {
      "expression": "author = :author",
        "expressionValues" : {
          ":author" : { "S" : "${context.arguments.author}" }
        }
    }
    #if( ${context.arguments.count} )
        ,"limit": ${context.arguments.count}
    #end
    #if( ${context.arguments.nextToken} )
        ,"nextToken": "${context.arguments.nextToken}"
    #end
}
```

  Note that 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).
- Paste the following into the "Configure the response mapping template" section:

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": "${context.result.nextToken}"
    #end
}
```

Note: This is the same response mapping template we used in the `allPost` resolver.

- Click the `Save` button.

See the Query (p. 175) reference documentation for more info about Query request mapping.

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. We can now query the table to retrieve all the posts by a specific author.

Before we do that, however, lets populate the table with some more posts, because every post so far has the same author.

- Go to the "Queries" tab
- Paste the following mutation into the "Queries" pane

```
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 }
}
```

- Then hit the "Execute query" button (the orange play button).

Now, lets query the table, returning all posts authored by Nadia.

- Paste the following query in the "Queries" pane

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Then hit the "Execute query" button (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 something like this:

```
{
  "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, lets look for all posts by AUTHORNAME, getting 5 at a time.

- Paste the following query in the "Queries" pane

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Then hit the "Execute query" button (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 something like this:

```
{
  "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":
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI3Z0lnSEx
    }
  }
}
```

- Update the `nextToken` argument with the value returned from the previous query:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
    nextToken:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI3Z0lnSEx
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Then hit the "Execute query" button (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 something like this:

```
{
  "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

So far our `Post` type has been a flat key/value object. It's also possible to model complex objects with the AWS AppSyncDynamoDB resolver, such as sets, lists, and maps.

Lets update our `Post` type to include tags. A post can have 0 or more tags, which are stored in DynamoDB as a String Set. We'll also set up some mutations to add and remove tags, and a new query to scan for posts with a specific tag.

- Go to the "Schema" tab
- Modify the `Post` type in the "Schema" pane to add a new `tags` field:

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
}
```

- Modify the `Query` type in the "Schema" pane to add a new `allPostsByTag` query:

```
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
}
```

- Modify the `Mutation` type in the "Schema" pane to add new `addTag` and `removeTag` mutations:

```
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!
}
```

- Click the `Save` button
- Find the newly created `allPostsByTag` field on the `Query` type in the "Data types" pane on the right.
- Click on its `Attach` button.
- Select `PostDynamoDBTable` in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "filter": {
      "expression": "contains (tags, :tag)",
        "expressionValues": {
          ":tag": { "S": "${context.arguments.tag}" }
        }
    }
    #if( ${context.arguments.count} )
        ,"limit": ${context.arguments.count}
    #end
    #if( ${context.arguments.nextToken} )
        ,"nextToken": "${context.arguments.nextToken}"
    #end
}
```

- Paste the following into the "Configure the response mapping template" section:

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": "${context.result.nextToken}"
    #end
}
```

- Click the `Save` button.
- Find the newly created `addTag` field on the `Mutation` type in the "Data types" pane on the right.
- Click on its `Attach` button.
- Select `PostDynamoDBTable` in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : { "S" : "${context.arguments.id}" }
    },
    "update" : {
        "expression" : "ADD tags :tags, version :plusOne",
        "expressionValues" : {
            ":tags" : { "SS": [ "${context.arguments.tag}" ] },
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- Paste the following into the "Configure the response mapping template" section:

```
$utils.toJson($context.result)
```

- Click the `Save` button.
- Find the newly created `removeTag` field on the `Mutation` type in the "Data types" pane on the right.
- Click on its `Attach` button.
- Select `PostDynamoDBTable` in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : { "S" : "${context.arguments.id}" }
    },
    "update" : {
        "expression" : "DELETE tags :tags ADD version :plusOne",
        "expressionValues" : {
            ":tags" : { "SS": [ "${context.arguments.tag}" ] },
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- Paste the following into the "Configure the response mapping template" section:

```
$utils.toJson($context.result)
```

- Click the `Save` button.

Call the API to work with tags

Now the resolvers have been set up, AWS AppSync knows how to translate incoming `addTag`, `removeTag`, and `allPostsByTag` requests into DynamoDB UpdateItem and Scan operations.

To try it out, lets select one of the posts we created earlier. For example, lets use one of Nadia's posts.

- Go to the "Queries" tab
- Paste the following query into the "Queries" pane.

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Then hit the "Execute query" button (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 something like this:

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

- Lets use the one with the title "The cutest dog in the world". Note down its `id` because we'll use it later.

Now let's try adding a "dog" tag.

- Paste the following mutation into the "Queries" pane. You'll also need to update the `id` argument to have the value we noted down earlier.

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

- Then hit the "Execute query" button (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"
      ]
    }
  }
}
```

We can add more tags as well.

- Update the mutation to change the `tag` argument to "puppy".

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- Then hit the "Execute query" button (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"
      ]
    }
  }
}
```

We can also delete tags:

- Paste the following mutation into the "Queries" pane. You'll also need to update the `id` argument to have the value we noted down earlier.

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- Then hit the "Execute query" button (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"
      ]
    }
  }
}
```

We can also search for all posts that have a tag:

- Paste the following query into the "Queries" pane.

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

- Then hit the "Execute query" button (the orange play button).
- All posts that have the "dog" tag are returned:

```
{
  "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, we can also use DynamoDB Lists and Maps to model complex data in a single object.

Lets add the ability to add comments to posts. This will be modeled as a List of Map objects on our Post object in DynamoDB.

Note: in a real application, we would model comments in their own table, however for the purpose of this tutorial we will just add them in the Post table.

- Go to the "Schema" tab
- Add a new `Comment` type in the "Schema" pane:

```
type Comment {
    author: String!
    comment: String!
}
```

- Modify the `Post` type in the "Schema" pane to add a new `comments` field:

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
  comments: [Comment!]
}
```

- Modify the `Mutation` type in the "Schema" pane to add a new `addComment` mutation:

```
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!
}
```

- Click the `Save` button
- Find the newly created `addComment` field on the `Mutation` type in the "Data types" pane on the right.

- Click on its `Attach` button.
- Select `PostDynamoDBTable` in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : { "S" : "${context.arguments.id}" }
    },
    "update" : {
        "expression" : "SET comments =
 list_append(if_not_exists(comments, :emptyList), :newComment) ADD version :plusOne",
        "expressionValues" : {
            ":emptyList": { "L" : [] },
            ":newComment" : { "L" : [
                { "M": {
                    "author": { "S" : "${context.arguments.author}" },
                    "comment": { "S" : "${context.arguments.comment}" }
                }}
            ] },
            ":plusOne" : { "N" : 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.
- Paste the following into the "Configure the response mapping template" section:

```
$utils.toJson($context.result)
```

- Click the `Save` button.

Call the API to add a comment

Now the resolvers have been set up, AWS AppSync knows how to translate incoming `addComment` requests into DynamoDB UpdateItem operations.

Lets try it out by adding a comment to the same post we added the tags to.

- Go to the "Queries" tab
- Paste the following query into the "Queries" pane.

```
mutation addComment {
  addComment(
    id:10
    author: "Steve"
    comment: "Such a cute dog."
  ) {
    id
    comments {
      author
      comment
    }
  }
}
```

- Then hit the "Execute query" button (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 something like this:

```
{
  "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 we've built an API that lets us manipualte Post objects in DynamoDB using AWS AppSync and GraphQL. For further information check out the Resolver Mapping Template Reference (p. 153).

To clean up, you can delete the AppSync GraphQL API from the console.

To delete the DynamoDB table and IAM role we created, 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
```

Tutorial: Lambda Resolvers

This is prerelease documentation for a service in preview release. It is subject to change.

AWS AppSync allows you to use AWS Lambda to resolve any GraphQL field. For example, a GraphQL query might call out to an Amazon RDS instance, and a GraphQL mutation might write to a Amazon Kinesis stream. This section outlines how you can 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 example.

```
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 handles retrieving a post by ID, adding a post, retrieving a list of posts, and fetching related posts for a given post.

Note: The `switch` statement on `event.field` allows the Lambda function to determine which field is being currently resolved.

Now let's create this Lambda function using the AWS console or with AWS CloudFormation by clicking here:

```
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 launch this AWS CloudFormation stack in the US West 2 (Oregon) region in your AWS account:

Configure data source for AWS Lambda

After the AWS Lambda function has been created, navigate to your AWS AppSync GraphQL API in the console and choose the **Data Sources** tab.

Select **New** and enter a friendly name for the data source, such as `"Lambda"`, and then select AWS Lambda for **Data source type**. Then choose the appropriate region. You should see your Lambda functions listed.

After selecting your Lambda function, you can either create a new role (for which AWS AppSync assigns the appropriate permissions) or choose an existing role that has the following inline policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:Invoke"
            ],
            "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/LAMBDA_FUNCTION"
        }
    ]
}
```

Creating a GraphQL Schema

Now that the data source is connected to your Lambda function, let's create a GraphQL schema.

From the schema editor in the AWS AppSync console, make sure you schema matches the schema below.

```
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]
}
```

Configuring resolvers

Now that we have registered a AWS Lambda data source and a valid GraphQL schema, we can connect our GraphQL fields to our Lambda data source using resolvers.

To create a resolver, we'll need mapping templates. To learn more about mapping templates, read AppSync mapping templates overview Resolver Mapping Template Overview (p. 153).

For more information about Lambda mapping templates, see Resolver Mapping Template Reference for Lambda (p. 199).

We are going to attach a resolver to our 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-hand side for `getPost(id:ID!): Post` click **Attach Resolver**.

Select your AWS Lambda data source and under the **request mapping template** section select the dropdown menu for **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)
    }
}
```

Now under the **response mapping template** section, select the drop-down menu for **Return Lambda Result**.

We will use the base template as-is. It should look like the following:

```
$utils.toJson($context.result)
```

Click **Save**. You have now attached your first resolver! Repeat this operation for the remaining fields as follows:

`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)
    }
}
```

`addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` response mapping template

```
$utils.toJson($context.result)
```

`allPosts: [Post]` request mapping template

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "allPosts"
    }
}
```

`allPosts: [Post]` response mapping template

```
$utils.toJson($context.result)
```

`Post.relatedPosts: [Post]` request mapping template

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "relatedPosts",
        "source":  $utils.toJson($context.source)
    }
}
```

`Post.relatedPosts: [Post]` response mapping template

```
$utils.toJson($context.result)
```

Testing 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.

In the AppSync console, on the left-hand side, choose the **Queries** tab. Populate it with 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(id: "2") {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

`allPosts` query

```
query {
    allPosts {
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {
            id
            title
        }
    }
}
```

Returning Errors

Any given field resolution can result in an error. AppSync lets you raise errors:

- From the request or response mapping template
- From the Lambda function

From the mapping template

The `$utils.error` helper method can be used from the VTL template to raise intentional errors. It takes as argument an `errorMessage`, an `errorType`, and an optional `data` value. The `data` comes handy for returning extra data back to the client, when an error has been raised. The `data` object will be added to the `errors` in the GraphQL final response.

For example using it in the `Post.relatedPosts: [Post]` response mapping template .. code-block:: sh

$utils.error("Failed to fetch relatedPosts", "LambdaFailure", $context.result)

would yield 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

AppSync also understands errors thrown from the Lambda function. The Lambda programming model lets you raise *Handled* errors. If an error is thrown from the Lambda function, AppSync will fail the resolution of the current field. Only the error message returned from Lambda will be set in the response. Also, it is currently impossible to 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, AppSync will use the error message set by AWS Lambda.

The following Lambda function raises an error:

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    callback("I fail. Always.");
};
```

Which would return a GraphQL response similar to below:

```
{
    "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

You may have noticed that the Lambda function in our example had a `relatedPosts` field which returned a list of related posts for a given post. In our example queries, the `allPosts` field invocation from our Lambda function returns 5 posts. Because we have specified that we also want to resolve `relatedPosts` for each returned post, the `relatedPosts` field operation will, in turn, be invoked 5 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 doesn't sound substantial for this specific use case, our application can get quickly undermined by this compounded over-fetching.

If, say, we 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 our Lambda function 1 + 5 + 25 = 31 times.

This is a fairly common challenge and is often called the N+1 problem, (in our case, N = 5) and it can incur increased latency and cost to our application.

One approach to solving this issue is to batch similar field resolver requests together. So in our example, instead of our Lambda function resolving a list of related posts for a single given post, it would instead resolve a list of related posts for a given batch of posts.

To see it in action, let's switch our `Post.relatedPosts: [Post]` resolver to a batch-enabled resolver.

In the AWS AppSync console, on the right-hand side, 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)
    }
}
```

Note that only the `operation` field has changed from `Invoke` to `BatchInvoke`. The payload field now becomes an array of whatever has been specified in the template, so in our example, our Lambda function will receive as input:

```
[
    {
        "field": "relatedPosts",
        "source": {
            "id": 1
        }
    },
    {
        "field": "relatedPosts",
        "source": {
            "id": 2
        }
    },
    ...
]
```

When `BatchInvoke` is specified in the request mapping template, the Lambda function is now given a list of requests and is also expected to return a list of results.

Specifically, the list of results *must* match in size and in order of the request payload entries, so AWS AppSync can match the results accordingly.

So in our example, because of batching, our Lambda function needs to return a batch of results:

```
[
    [{"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 AWS Lambda function in Node.js demonstrates this batching functionality for the `Post.relatedPosts` field:

```
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

We saw previously that it is 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 will flag an entire batch as failed. This might be fine for specific scenarios where an irrecoverable error happened, such as, the connection to a data store going down. However, in cases where some items in the batch succeed, and some others fail, let see how it is possible to return both errors and valid data. AppSync only imposes the batch response to be a list of elements matching the original size of the batch, it is up to you to define a data structure that can differentiate valid data from an error.

For instance, if our Lambda function is expected to return a batch of related posts, we could instead return a list of `Response` object where each object has optional *data*, *errorMessage* and *errorType* fields. If the *errorMessage* field is present, it means there was an error.

See below the updated 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;
    }
};
```

And we could write a response mapping template to parse each item of our Lambda function, and raise an error if needed:

```
#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 would return a GraphQL response similar to below:

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

Tutorial: Amazon Elasticsearch Service Resolvers

This is prerelease documentation for a service in preview release. It is subject to change.

AWS AppSync supports using Amazon Elasticsearch Service from domains that you have provisioned in your own AWS account. 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 Elasticsearch (p. 196).

Create a New Amazon ES Domain

To get started with this tutorial, you need an existing Amazon ES domain. If you don't have one, you can use the following sample. Note that it can take up to 15 minutes for an Amazon ES domain to be created before you can move on to integrating it with an AWS AppSync data source.

```
aws cloudformation create-stack --stack-name DDElasticsearch \
--template-url https://s3-us-west-2.amazonaws.com/awsappsync/resources/elasticsearch/
ESResolverCFTemplate.yaml \
--parameters ParameterKey=ESDomainName,ParameterValue=ddtestdomain
 ParameterKey=Tier,ParameterValue=development \
--capabilities CAPABILITY_NAMED_IAM
```

You can launch this AWS CloudFormation stack in the US West 2 (Oregon) region in your AWS account:

Launch Stack

Configure Data Source for Amazon ES

After the Amazon ES 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 "Elasticsearch". Then choose **Amazon Elasticsearch cluster** for **Data source type**, choose the appropriate region, and you should see your Amazon ES 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:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1234234",
            "Effect": "Allow",
            "Action": [
                "es:ESHttpDelete",
                "es:ESHttpHead",
                "es:ESHttpGet",
                "es:ESHttpPost",
                "es:ESHttpPut"
            ],
            "Resource": [
                "arn:aws:es:REGION:ACCOUNTNUMBER:domain/democluster/*"
            ]
        }
    ]
}
```

Connecting a Resolver

Now that the data source is connected to your Amazon ES 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): Post
 }

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 Amazon ES domain, which would map to a path root of `/id/post`, where `id` is the index and `post` is the type. From this root path, you can perform individual document searches, wildcard searches with `/id/post*` or multi-document searches with a **path** of `/id/post/_search`. If you have another type `User`, for example, one that is indexed under the same index `id`, you can perform multi-document searches with a **path** of `/id/_search`. This searches for fields on both `Post` and `User`.

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**. Choose your Amazon ES 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 `/id/post/_search`. It should look like the following:

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/id/post/_search",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "from":0,
            "size":50
        }
    }
}
```

This assumes that the preceding schema has documents with an `id` field, and that the documents have been indexed in Amazon ES by this 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 Amazon ES 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 Amazon ES data source, but use the following example in the **response mapping template**:

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/id/post/_search",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "from":0,
            "size":50,
            "query":{
                "term" :{
                    "author":"$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 Amazon ES

You may want to add data to your Amazon ES 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 Amazon ES domain.

Return to the **Schema** page in the AWS AppSync console and select **Attach resolver** for the `addPost()` mutation. Select the Amazon ES data source again and use the following **response mapping template** for the `Posts` schema:

```
{
    "version":"2017-02-28",
    "operation":"PUT",
    "path":"/id/post/$context.arguments.id",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "id":"$context.arguments.id",
            "author":"$context.arguments.author",
            "ups":"$context.arguments.ups",
            "downs":"$context.arguments.downs",
            "url":"$context.arguments.url",
            "content":"$context.arguments.content",
            "title":"$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 be explained more in the next section:

```
$utils.toJson($context.result.get("_source"))
```

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 Amazon ES data source again and use the following mapping template:

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/id/post/$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 Amazon ES domain. Navigate to the **Queries** tab of the AWS AppSync console and add a new record:

```
mutation {
    addPost(
        id:"12345"
        author: "Fred"
        title: "My first book"
        content: "This will be fun to write!"
    ){
        id
        author
        title
    }
}
```

If the record is inserted successfully, you'll see the fields on the right. Similarly, you can now run a `searchPosts` query against your Amazon ES domain:

```
query {
    searchPosts {
        id
        title
        author
        content
    }
}
```

Best Practices

- Amazon ES should be for querying data, not as your primary database. You may want to use Amazon ES 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.

Tutorial: Local Resolvers

This is prerelease documentation for a service in preview release. It is subject to change.

AWS AppSync allows you to use supported data sources (AWS Lambda, Amazon DynamoDB, or Amazon Elasticsearch 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 (p. 138) 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": "${context.arguments.body}",
    "from": "${context.identity.username}",
    "to":  "${context.arguments.to}",
    "sentAt": "$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.

Tutorial: Combining GraphQL Resolvers

This is prerelease documentation for a service in preview release. It is subject to change.

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 show how you might mix and match data sources in your schema, but before doing so you should have familiarity with setting up data sources and resolvers for AWS Lambda, Amazon DynamoDB and Amazon Elasticsearch Service as outlined in the previous sections.

Example Schema

The below 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 (p. 166). However, there are alternatives to meet your business needs, such as having these GraphQL queries resolve from AWS Lambda or Amazon ES.

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 compatability issues. For illustrative purposes in the below example, we show an AWS Lambda function that 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 Amazon ES

For some applications, you might perform mutations or simple lookup queries against DynamoDB, and have a background process transfer documents to Amazon ES. You can then simply attach the `searchPosts` Resolver to the Amazon ES 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. Transfering data from DynamoDB could be done through an ETL process or alternatively you can stream from DynamoDB using Lambda with the following example code:

Note: This code is for example only.

```
var AWS = require('aws-sdk');
var path = require('path');
var stream = require('stream');

var esDomain = {
    endpoint: 'https://elasticsearch-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 Amazon ES domain. For more information on configuring this, see the DynamoDB Streams documentation.

Real-Time Data

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. Subscription connection management is handled automatically by the AWS AppSync client SDK using MQTT over Websockets as the network protocol between the client and service.

Note: Adding resolvers to subscriptions is unsupported at this time.

Subscriptions are triggered from mutations and the mutation selection set is sent to subscribers.

The following example shows how to work with GraphQL subscriptions. Notice that it doesn't specify a data source, because the data source could be AWS Lambda, Amazon DynamoDB, or Amazon Elasticsearch Service.

To get started to with subscriptions, you must add a subscription entry point to your schema:

```
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. You would 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 trigger 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
    }
}
```

Although the subscription query above is needed for client connections and tooling, the selection set that is received by subscribers is specified by the client triggering the mutation. To demonstrate this, if a mutation was made from another mobile client or a server, for example, `mutation addPost(...){id author title })`, then content, version and url wouldn't be published to subscribers. Instead id, author and title would be published.

In the example above, we showed subscriptions without arguments. If your schema looked like this:

```
type Subscription {
    updatedPost(id:ID! author:String): Post
    @aws_subscribe(mutations: ["updatePost"])
}
```

then your client would define a subscription:

```
subscription UpdatedPostSub {
    updatedPost(id:"XYZ", author:"ABC") {
        title
        content
    }
}
```

One final thing to note: 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 App (p. 31).

Using Subscription Arguments

An important part of using GraphQL subscriptions is understanding when and how to use arguments, as subtle changes will allow you to modify how and when clients are notified of mutations that have occured. To do this, refer to the sample schema from the Quickstart section, which creates "Events" and "Comments". For the sample schema, you will see the following mutation:

```
type Mutation {
    createEvent(
            name: String!,
                when: String!,
                where: String!,
                description: String!
        ): Event
    deleteEvent(id: ID!): Event
        commentOnEvent(eventId: ID!, content: String!, createdAt: String!): Comment
}
```

In the default sample, clients can subscribe to Comments when a specific eventId argument is passed through:

```
type Subscription {
    subscribeToEventComments(eventId: String!): Comment
        @aws_subscribe(mutations: ["commentOnEvent"])
}
```

However, if you want to allow clients to subscribe to a single event OR all events, you can make this argument optional by removing the exclamation point (!) from the subscription prototype:

```
subscribeToEventComments(eventId: String): Comment
```

With this change, clients that omitted this argument would get comments for all events. Additionally if you wanted clients to explicitly subscribe to all comments for all events, you would remove the argument:

```
subscribeToEventComments(): Comment
```

These are for comments on one or more events. If you just wanted to know about all events that get created, you might do something like this:

```
type Subscription {
    subscribeToNewEvents(): Event
        @aws_subscribe(mutations: ["createEvent"])
}
```

Multiple arguments can also be passed. For example, if you want to get notified of new events at a specific place and time:

```
type Subscription {
    subscribePlaceDate(where: String! when: String!): Event
            @aws_subscribe(mutations: ["createEvent"])
}
```

The client application could now do this:

```
subscription myplaces {
    subscribePlaceDate(where: "Seattle" when: "Saturday"){
        id
        name
        description
    }
}
```

Security

This is prerelease documentation for a service in preview release. It is subject to change.

Topics

This section describes options for configuring security and data protection for your applications.

There are three 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_IAM`**

  For using AWS Identity and Access Management (IAM) permissions.
- **`AMAZON_COGNITO_USER_POOLS`**

  For using an Amazon Cognito user pool.

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.

API keys are configurable for upto 365 days, and you can extend an existing expiration date for upto 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 } }" }' http://YOURAPPSYNCENDPOINT/graphql
```

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:

```
{
   "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.

```
{
   "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:

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
         "appsync:GraphQL"
         ],
         "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Query/
fields/posts"
         ]
      }
      ]
   }
   ]
}
```

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 your user pools 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"}'
```

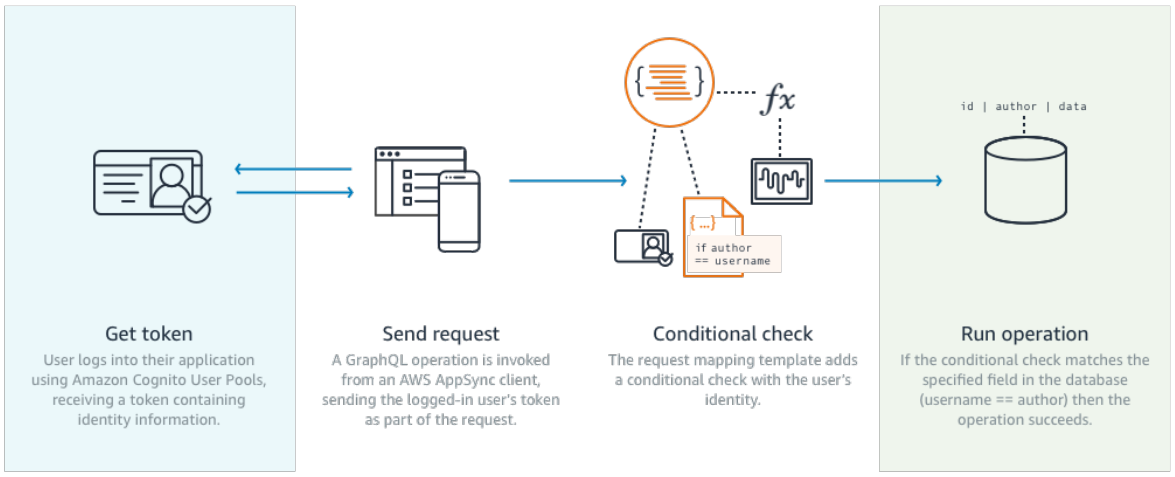
Fine-Grained Access Control

The preceding information demonstrates how to restrict or grant access to certain GraphQL fields. If you want to set access controls on the data itself based on certain conditions - such as who the user is that is making a call and whether they own the data - you can do use mapping templates in your resolvers. You can also perform more complex business logic, which we describe in Filtering Information (p. 146).

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 (p. 153) and the Resolver Mapping Template Reference for DynamoDB (p. 166).

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.


id | author | data
if author == username
Get token
User logs into their application using Amazon Cognito User Pools, receiving a token containing identity information.
Send request
A GraphQL operation is invoked from an AWS AppSync client, sending the logged-in user's token as part of the request.
Conditional check
The request mapping template adds a conditional check with the user's identity.
Run operation
If the conditional check matches the specified field in the database (username == author) then the operation succeeds.

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 DynamoDB`UpdateItem` 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 request mapping template for `addPost` that stores the user identity as an `Author` column:

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "postId" : { "S" : "${context.arguments.id}" }
    },
    "attributeValues" : {
        "Author" : {"S" : "${context.identity.user}"}
        #foreach( $entry in $context.arguments.entrySet() )
            #if( $entry.key != "id" )
                ,"${entry.key}" : { "S" : "${entry.value}" }
            #end
        #end
    },
    "condition" : {
        "expression" : "attribute_not_exists(postId)"
    }
}
```

Note that the `Author` attribute is populated from the `Identity` object, which came from the application.

Finally, here is an example of the request mapping template for `editPost`, which only updates the content of the blog post if the request comes from the user that created the post:

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "postId" : { "S" : "${context.arguments.id}" }
    },
    "attributeValues" : {
        "Author" : {"S" : "${context.identity.user}"}
        #foreach( $entry in $context.arguments.entrySet() )
            ,"${entry.key}" : { "S" : "${entry.value}" }
        #end
    },
    "condition" : {
        "expression"       : "Author = :authorName",
        "expressionValues" : {
            ":authorName"      : { "S" : "${context.identity.user}" }
        }
    }
}
```

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 run a `GetItem` query with the following mapping template:

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "postId" : { "S" : "${context.arguments.id}" }
    }
}
```

This returns all the values responses, even if the caller isn't the author who created the post. To prevent this from happening, you can perform the access check on the response mapping template in this case as follows:

```
{
   #if($context.result["Author"] == "$context.identity.user")
      $utils.toJson($context.result);
   #end
}
```

If the caller doesn't match this check, only a null response is returned.

Authorization Use Cases

This is prerelease documentation for a service in preview release. It is subject to change.

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 AppSync Resolvers, please 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.

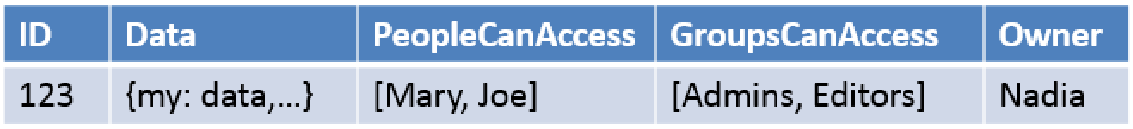
AWS AppSync uses resources in your own account and threads identity (user/role) information into the GraphQL request and response as a context object which you can use in the resolver. This means that permissions can be granted appropriately either on write or read operations based on the resolver logic. 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 DynamoDB resolver reference.

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 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 Cognito User Pools is used and will be the user identity if AWS IAM is used (including Cognito Federated Identities). There are other values to store for an owner such as the unique "Cognito identity" value, which is useful when federating logins from multiple locations, and you should review the options available in <resolver-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 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" : { "S" : "${context.argument.id}" }
    },
    "attributeValues" : {
        "owner" : {"S" : "${context.identity.username}"}
        #foreach( $entry in $context.arguments.entrySet() )
            ,"${entry.key}" : { "S" : "${entry.value}" }
        #end
    }
}
#else
    $utils.unauthorized()
#end
```

Use Case: Filtering a list of results

In the above 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 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(${context.arguments.count}) "${context.arguments.count}" #else 20 #end,
    "nextToken": #if(${context.arguments.nextToken}) "${context.arguments.nextToken}" #else
 null #end,
    "filter":{
        "expression": "contains(#peopleCanAccess, :value)",
        "expressionNames": {
                "#peopleCanAccess": "peopleCanAccess"
        },
        "expressionValues": {
                ":value": { "S" : "${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(${context.arguments.count}) "${context.arguments.count}" #else 20 #end,
    "nextToken": #if(${context.arguments.nextToken}) "${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 the Security section an example was given 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" : { "S" : "${context.arguments.id}" }
    },
    "update" : {
        "expression" : "SET meta = :meta",
        "expressionValues": {
            ":meta" : { "S": "${context.arguments.meta}" }
        }
    },
    "condition" : {
        "expression"       : "contains(Owner,:expectedOwner)",
        "expressionValues" : {
            ":expectedOwner" : { "S" : "${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" : { "S" : "$context.arguments.id" }
        ## If your table has a sort key, add it as an item here. **
    },
    "attributeValues" : {
        ## Add an item for each field you would like to store to Amazon DynamoDB. **
        "title" : { "S" : "${context.arguments.title}" },
        "content": { "S" : "${context.arguments.content}" },
        "owner": {"S": "${context.identity.username}" }
    },
    "condition" : {
        "expression": "attribute_not_exists(id) OR $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" : { "S" : "${context.arguments.id}" }
    },
    "update":{
                "expression" : "SET title = :title, content = :content",
        "expressionValues": {
            ":title" : { "S": "${context.arguments.title}" },
            ":content" : { "S": "${context.arguments.content}" }
        }
    },
    "condition" : {
        "expression": "$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` whith 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.

Resolver Mapping Template Reference

This is prerelease documentation for a service in preview release. It is subject to change.

Topics

Resolver Mapping Template Overview

This is prerelease documentation for a service in preview release. It is subject to change.

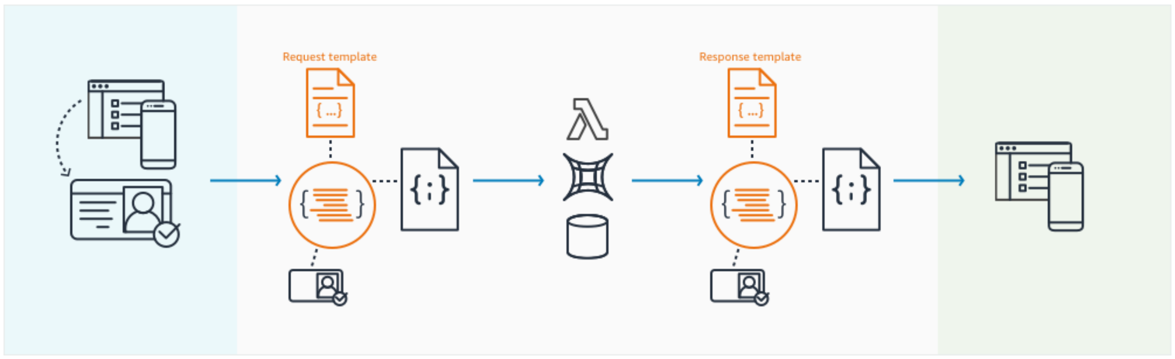
AWS AppSync lets you respond to GraphQL operations by enabling you to perform operations on your AWS resources. For each data source, a GraphQL resolver must run and be able to communicate with that data source appropriately.

Usually, the communication is through parameters or operations that are unique to the data source. For an AWS Lambda resolver, you need to specify the payload. For an Amazon DynamoDB resolver, you need to specify a key. For an Amazon Elasticsearch Service resolver, you need to specify an index and the query operation.

Mapping templates are a way of indicating to 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 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 main types of mapping templates:

- Request templates: Take the incoming request after a GraphQL operation is parsed and convert it into instructions for the resolver so that the resolver can call your data source.
- Response templates: Interpret responses from your data source and translate into a GraphQL response, optionally performing some logic or formatting first.



Example Template

For example, suppose you have a DynamoDB data source and a 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" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "${context.arguments.id}" }
    }
}
```

This would substitute the `id` input parameter value of `1` for `${context.arguments.id}` and generate the following JSON:

```
     {
    "version" : "2017-02-28",
    "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" : ${context.data.id},
        "title" : "${context.data.theTitle}",
        "content" : "${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:

```
$utils.toJson($context.result)
```

Resolver Mapping Template Programming Guide

This is prerelease documentation for a service in preview release. It is subject to change.

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 logicful 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. All of these things are 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 always write Lambda functions in your 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 supress this behavior. The syntax for this is `$!{}`. For example, if we changed the previous code slightly to use `$!{somethingelse}`, the printing would be supressed:

```
#set($myValue = 5)
##Prints '5'
$myValue

##Nothing prints out
$!{somethingelse}
```

Calling Methods

You saw one example earlier of creating a variable and simultaneously setting values. You could also do this in two steps by adding data to your map:

```
#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 WILL NOT supress 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 will add 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" : "${context.arguments.author}"}` when using arguments from GraphQL clients, or for automatic ID generation like `"id" : { "S" : "$utils.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
<math>arr2[</math>
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 an 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))
```

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 autopopulating 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") and default ownership
```

```
#set($myMap.defaultOwners = ["Admins", "Editors"]``
```

Context

Now that you are more familiar with performing logical checks in AWS 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)
    $utils.toJson($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.

Appendix - Template Sample

If you followed along with the tutorial, you may have built out this template step by step. However, but we also include it below to copy/paste for your 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

##Operatorsdoesn'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)
    $utils.toJson($context.result);
  #else
    $util.unauthorized()
#end
```

Resolver Mapping Template Reference for DynamoDB

This is prerelease documentation for a service in preview release. It is subject to change.

The AWS AppSyncDynamoDB resolver enables you to use GraphQL to store and retrieve data in existing Amazon DynamoDB tables in your account. This resolver works by enabling you to map an incoming GraphQL request into a DynamoDB call, and then map the DynamoDB response back to GraphQL. This section describes the mapping templates for supported DynamoDB operations.

Topics

GetItem

The `GetItem` request mapping document lets you tell the AWS AppSyncDynamoDB resolver to make a `GetItem` request to DynamoDB, and allows 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
}
```

The fields are defined as follows:

`version`

The template definition version. Only `2017-02-28` is 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 180). This value is required.

`consistentRead`

Whether or not to perform a strongly consistent read with DynamoDB. This is optional, and defaults to `false`.

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) (p. 184).

For more information about response mapping templates, see Resolver Mapping Template Overview (p. 153).

Example

Following is a mapping template for a GraphQL query `getThing(foo: String!, bar: String!)`:

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : { "S" : "${context.arguments.foo}" },
        "bar" : { "S" : "${context.arguments.bar}" }
    },
    "consistentRead" : true
}
```

See the DynamoDB API documentation for more information about the DynamoDB`GetItem` API.

PutItem

The `PutItem` request mapping document lets you tell the AWS AppSyncDynamoDB resolver to make a `PutItem` request to DynamoDB, and allows 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" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "attributeValues" : {
        "baz" : ... typed value
    },
    "condition" : {
       ...
    }
}
```

The fields are defined as follows:

`version`

The template definition version. Only `2017-02-28` is 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 180). This value is required.

`attributeValues`

The rest of the attributes of the item to be put into DynamoDB. For more information on how to specify a "typed value", see Type System (Request Mapping) (p. 180). 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 will overwrite any existing entry for that item. For more information on conditions, see Condition Expressions (p. 188). This value 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) (p. 184).

For more information about response mapping templates, see Resolver Mapping Template Overview (p. 153).

Example 1

Following 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 will be created. If an item already exists with the specified key, it will be overwritten.

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "foo" : { "S" : "${context.arguments.foo}" },
        "bar" : { "S" : "${context.arguments.bar}" }
    },
    "attributeValues" : {
        "name"    : { "S" : "${context.arguments.name}" },
        "version" : { "N" : ${context.arguments.version} }
    }
}
```

Example 2

Following 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" : { "S" : "${context.arguments.foo}" },
        "bar" : { "S" : "${context.arguments.bar}" }
    },
    "attributeValues" : {
        "name"    : { "S" : "${context.arguments.name}" },
        #set( $newVersion = $context.arguments.expectedVersion + 1 )
        "version" : { "N" : ${newVersion} }
    },
    "condition" : {
        "expression" : "version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" : { "N" : ${context.arguments.expectedVersion} }
        }
    }
}
```

See the DynamoDB API documentation for more information about the DynamoDB`PutItem` API.

UpdateItem

The `UpdateItem` request mapping document lets you tell the AWS AppSyncDynamoDB resolver to make an `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" : "2017-02-28",
    "operation" : "UpdateItem",
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "update" : {
        "expression" : "someExpression"
        "expressionNames" : {
           "#foo" : "foo"
       },
       "expressionValues" : {
           ":bar" : ... typed value
       }
    },
    "condition" : {
        ...
    }
}
```

The fields are defined as follows:

`version`

The template definition version. Only `2017-02-28` is 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 180). This value is required.

`update`

The `update` section lets you specify an update expression that describes how to update the item in DynamoDB. See the DynamoDB UpdateExpressions documentation for more information on how to write update expressions. 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 180). 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 will update any existing entry regardless of its current state. For more information on conditions, see Condition Expressions (p. 188). This value 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) (p. 184).

For more information about response mapping templates, see Resolver Mapping Template Overview (p. 153).

Example 1

Following 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" : { "S" : "${context.arguments.id}" }
    },
    "update" : {
        "expression" : "ADD #votefield :plusOne, version :plusOne",
        "expressionNames" : {
            "#votefield" : "upvotes"
        },
        "expressionValues" : {
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

Example 2

Following 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 checks to be sure the item currently in DynamoDB has the `version` field set to `expectedVersion`:

```
{
    "version" : "2017-02-28",

    "operation" : "UpdateItem",

    "key" : {
        "id" : { "S" : "${context.arguments.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" : { "N" : ${context.arguments.expectedVersion} }
        }
    }
}
```

See the DynamoDB API documentation for more information about the DynamoDB`UpdateItem` API.

DeleteItem

The `DeleteItem` request mapping document lets you tell the AWS AppSyncDynamoDB resolver to make a `DeleteItem` request to DynamoDB, and allows 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" : "2017-02-28",
    "operation" : "DeleteItem",
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "condition" : {
        ...
    }
}
```

The fields are defined as follows:

`version`

The template definition version. Only `2017-02-28` is 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 180). 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 will delete an item regardless of its current state. For more information on conditions, see Condition Expressions (p. 188). This value 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) (p. 184).

For more information about response mapping templates, see Resolver Mapping Template Overview (p. 153).

Example 1

Following is a mapping template for a GraphQL mutation `deleteItem(id: ID!)`. If an item exists with this ID, it will be deleted.

```
{
    "version" : "2017-02-28",
    "operation" : "DeleteItem",
    "key" : {
        "id" : { "S" : "${context.arguments.id}" }
    }
}
```

Example 2

Following is a mapping template for a GraphQL mutation `deleteItem(id: ID!, expectedVersion: Int!)`. If an item exists with this ID, it will be deleted, but only if its `version` field set to `expectedVersion`:

```
{
    "version" : "2017-02-28",
    "operation" : "DeleteItem",
    "key" : {
        "id" : { "S" : "${context.arguments.id}" }
    },
    "condition" : {
        "expression"       : "attribute_not_exists(id) OR version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" : { "N" : ${context.arguments.expectedVersion} }
        }
    }
}
```

See the DynamoDB API documentation for more information about the DynamoDB`DeleteItem` API.

Query

The `Query` request mapping document lets you tell the AWS AppSyncDynamoDB resolver to make a `Query` request to DynamoDB, and allows 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",
    "filter" : {
        ...
    }
}
```

The fields are defined as follows:

version

The template definition version. Only `2017-02-28` is 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. See the DynamoDB KeyConditions documentation for more information on how to write key condition expressions. 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 180). 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 on filters, see Filters (p. 187). 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 will tell DynamoDB to query the specified index. If omitted, the primary key index will be 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 results to fetch at a single time. 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 AppSyncDynamoDB resolver will only return 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, then for each matching item in the index DynamoDB will fetch the entire item from the parent table. 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.

`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`.

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) (p. 184).

For more information about response mapping templates, see Resolver Mapping Template Overview (p. 153).

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` will contain a pagination token that can be used in another request. Note that AWS AppSync will encrypt and obfuscate the pagination token returned from DynamoDB. This is so data from your tables are not 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

Following 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" : { "S" : "${context.arguments.owner}" }
        }
    }
    "index" : "owner-index"
}
```

See the DynamoDB API documentation for more information about the DynamoDB `Query` API.

Scan

The `Scan` request mapping document lets you tell the AWS AppSyncDynamoDB resolver to make a `Scan` request to DynamoDB, and allows 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" : {
        ...
    }
}
```

The fields are defined as follows:

`version`

The template definition version. Only `2017-02-28` is 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`

An filter that can be used to filter the results from DynamoDB before they are returned. For more information on filters, see Filters (p. 187). 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, then the primary key index will be queried.

limit

The maximum number of results to fetch at a single time. This field is optional.

consistentRead

A boolean indicating 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 AppSyncDynamoDB resolver will only return 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, then for each matching item in the index DynamoDB will fetch the entire item from the parent table. If the index is configured to project all item attributes, then all of the data can be obtained from the local secondary index, and no fetching is required.

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`.

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.

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) (p. 184).

For more information about response mapping templates, see Resolver Mapping Template Overview (p. 153).

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` will contain a pagination token that can be used in another request. Note that AWS AppSync will encrypt and obfuscate the pagination token returned from DynamoDB. This is so data from your tables are not inadvertently leaked to the caller. Also note that these pagination tokens cannot 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

Following 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

Following 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" : { "S" : "${context.arguments.title}" }
        },
    }
}
```

See the DynamoDB API documentation for more information about the DynamoDB `Scan` API.

Type System (Request Mapping)

When using the AWS AppSyncDynamoDB 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 DynamoDBData 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 will not be parsed.

A DynamoDB value is used anywhere in a request mapping document where you need to specify a value. Some places where you will 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 will decode 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" : {
   "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 will decode 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 }
}
```

See the DynamoDB documentation for more information on each type.

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 AppSyncDynamoDB resolver converts it into GraphQL and JSON types as:

```
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

This section explains how AWS AppSync will convert the following DynamoDB scalar, document and set types:

String type `S`

A single string value. A DynamoDB String value will be returned simply as a string.

For example, if DynamoDB returned the following DynamoDB String value:

```
{ "S" : "some string" }
```

AWS AppSync will convert it to a string:

```
"some string"
```

String set type `SS`

A set of string values. A DynamoDB String Set value will be returned as a list of strings.

For example, if DynamoDB returned the following DynamoDB String Set value:

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync will convert 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 will be returned as a number.

For example, if DynamoDB returned the following DynamoDB Number value:

```
{ "N" : 1234 }
```

AWS AppSync will convert it to a number:

```
1234
```

Number set type `NS`

A set of number values. A DynamoDB Number Set value will be 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 will convert it to a list of numbers:

```
[ 67.8, 12.2, 70 ]
```

Binary type `B`

A binary value. A DynamoDB Binary value will be 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 will convert 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 will be 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 will convert 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 will be returned as a boolean.

For example, if DynamoDB returned the following DynamoDB Boolean value:

```
{ "BOOL" : true }
```

AWS AppSync will convert it to a boolean:

```
true
```

List type `L`

A list of any other supported DynamoDB value. A DynamoDB List value will be 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 will convert 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 will be 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 will convert 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 will convert 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. See the DynamoDB QueryFilter and DynamoDB ScanFilter documentation for more information on how to write filter expressions. 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 180). 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

Following 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" : { "S" : "${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 AppSyncDynamoDB 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 does not have a condition expression, so it will put an item in DynamoDB even if an item with the same key already exists, 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 will only let the operation succeed 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, then the AWS AppSyncDynamoDB resolver will return an error for the mutation and the current value of the object in DynamoDB in a `data` field in the `error` section of the GraphQL response. However, the AWS AppSyncDynamoDB resolver offers some additional features to help developers handle some common edge cases:

- If AWS AppSyncDynamoDB resolver can determine that the current value in DynamoDB matches the desired result, then it will treat 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 AppSyncDynamoDB resolver should handle the failure.

These will be described in greater detail in the Handling a Condition Check Failure (p. 190) section.

See the DynamoDB ConditionExpressions documentation for more information about DynamoDB conditions expressions.

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. See the DynamoDB ConditionExpressions documentation for more information about how to write condition expressions. 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 on 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 AppSyncDynamoDB resolver how to handle a condition check failure:

`equalsIgnore`

When a condition check fails when using the `PutItem` operation, the AWS AppSyncDynamoDB resolver will compare the item currently in DynamoDB against the item it tried to write. If they are the same, then it will treat 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, then treat the operation as it if succeeded. This field is optional.

`consistentRead`

When a condition check fails, AWS AppSync will get the current value of the item from DynamoDB using a strongly consistent read. You can use this field to tell the AWS AppSyncDynamoDB 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 AppSyncDynamoDB resolver will treat 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 AppSyncDynamoDB resolver will take after it has compared the current value in DynamoDB against the expected result. This field is required, and has two possible values:

`Reject`

The mutation will fail, and an error for the mutation and the current value of the object in DynamoDB in a `data` field in the `error` section of the GraphQL response.

`Custom`

The AWS AppSyncDynamoDB resolver will invoke 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 to decide how the AWS AppSyncDynamoDB resolver should handle the condition check failure. This field must only be specified when `strategy` is set to `Custom`. See Handling a Condition Check Failure (p. 190) for more information about how to use this feature.

Handling a Condition Check Failure

By default, when a condition check fails, the AWS AppSyncDynamoDB resolver will return an error for the mutation and the current value of the object in DynamoDB in a `data` field in the `error` section of the GraphQL response. However, the AWS AppSyncDynamoDB resolver offers some additional features to help developers handle some common edge cases:

- If AWS AppSyncDynamoDB resolver can determine that the current value in DynamoDB matches the desired result, then it will treat 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 AppSyncDynamoDB resolver should handle the failure.

The flowchart for this process is:

Checking for the Desired Result

When the condition check fails, the AWS AppSyncDynamoDB resolver will perform a `GetItem`DynamoDB request to get the current value of the item from DynamoDB. By default, it will use 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 AppSyncDynamoDB resolver will compare 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, then it will treat the operation as successful and return the item that was retrieved from DynamoDB. Otherwise, it will then follow the configured strategy.

  For example, if the `PutItem` request mapping document looked like this:

  ```
  {
     "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 this:

  ```
  {
     "id" : { "S" : "1" },
     "name" : { "S" : "Steve" },
     "version" : { "N" : 8 }
  }
  ```

  Then the AWS AppSyncDynamoDB 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 AppSyncDynamoDB resolver will see if an item was returned from DynamoDB. If no item was returned, it will treat the operation as successful. Otherwise, it will follow the configured strategy.
- For the `UpdateItem` operation, the AWS AppSyncDynamoDB 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, then the AWS AppSyncDynamoDB resolver will follow the configured strategy, to either reject the mutation or invoke a Lambda function to decide what to do next.

Following the "Reject" Strategy

When following the `Reject` strategy, the AWS AppSyncDynamoDB resolver will return an error for the mutation, and the current value of the object in DynamoDB will also be returned in a `data` field in the `error` section of the GraphQL response. The item returned from DynamoDB will be put through the response mapping template to translate it into a format the client expects, and will also be filtered by the selection set.

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:

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

and the response mapping template looks like:

```
{
   "id" : "${context.result.id}",
   "Name" : "${context.result.name}",
   "theVersion" : ${context.result.version}
}
```

then the GraphQL response will look like:

```
{
  "data": null,
  "errors": [
    {
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "data": {
        "Name": "Steve",
        "theVersion": 8
      },
      ...
    }
  ]
}
```

Also note that if any fields in the returned object would have been filled by other resolvers if the mutation had succeeded, they will not be resolved when the object is returned in the `error` section.

Following the "Custom" Strategy

When following the `Custom` strategy, the AWS AppSyncDynamoDB resolver will invoke a Lambda function to decide what to do next. The Lambda function has three options to choose from:

- `reject` the mutation. This will tell the AWS AppSyncDynamoDB 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. This will tell the AWS AppSyncDynamoDB resolver to silently ignore the condition check failure, and just return the value in DynamoDB.
- `retry` the mutation. This will tell the AWS AppSyncDynamoDB resolver to retry the mutation with a new request mapping document.

The Lambda invocation request

The AWS AppSyncDynamoDB resolver will invoke the Lambda function specified in the `lambdaArn`. It will use 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": "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 AppSyncDynamoDB 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 will tell the AWS AppSyncDynamoDB 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 will tell the AWS AppSyncDynamoDB resolver to silently ignore the condition check failure, and just return the value in DynamoDB.
- `retry` the mutation. The response payload for this option must have this structure:

  ```
  {
      "action": "retry",
      "retryMapping": { ... }
  }
  ```

  This will tell the AWS AppSyncDynamoDB 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. See PutItem (p. 168) for a description of the `attributeValues` field.

  ```
  {
  ```

```
    "attributeValues": { ... },
    "condition": {
        "equalsIgnore" = [ ... ],
        "consistentRead" = true
    }
}
```

For `UpdateItem`, the `retryMapping` section has the following structure. See UpdateItem (p. 170) for a description of the `update` section.

```
{
    "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
    }
}
```

Note that there is no way to specify a different operation or key to work on: the AWS AppSyncDynamoDB resolver will only allow retries of the same operation on the same object. Also note the `condition` section doesn't allow a `conditionalCheckFailedHandler` to be specified. If the retry fails, then the AWS AppSyncDynamoDB resolver will follow 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` then it will retry the request, updating the `version` and `expectedVersion` from the item currently in DynamoDB; otherwise it will reject 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)
};
```

Resolver Mapping Template Reference for Elasticsearch

This is prerelease documentation for a service in preview release. It is subject to change.

The AWS AppSync resolver for Amazon Elasticsearch Service enables you to use GraphQL to store and retrieve data in existing Amazon ES domains in your account. This resolver works by allowing you to map an incoming GraphQL request into an Amazon ES request, and then map the Amazon ES response back to GraphQL. This section describes the mapping templates for the supported Amazon ES operations.

Request Mapping Template

Most Amazon ES request mapping templates have a common structure where just a few pieces change. The following example runs a search against an Amazon ES 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:

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/id/post/_search",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "from":0,
            "size":50,
            "query" : {
                "bool" : {
                    "should" : [
                        {"match" : { "author" : "Nadia" }},
                        {"match" : { "author" : "Bailey" }}
                    ]
                }
            }
        }
    }
}
```

For more information on query options, see the Elasticsearch Query DSL Reference.

Response Mapping Template

As with other data sources, Amazon ES sends a response to AWS AppSync that needs to be converted to GraphQL. The shape of an Amazon ES response can be seen in the Elasticsearch Request Body Search DSL Reference.

Most GraphQL queries are looking for the `_source` field from an Amazon ES 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 Amazon ES:

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

(REQUEST Mapping Template only)

HTTP method or verb (GET, POST, PUT, HEAD or DELETE) that AWS AppSync sends to the Amazon ES domain. Both the key and the value must be a string.

```
"operation" : "PUT"
```

`path` field

(REQUEST Mapping Template only)

The search path for an Amazon ES 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 mapping template is evaluated, this path is sent as part of the HTTP request, including the Amazon ES domain. For example, the previous example might translate to:

```
GET https://elasticsearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

`params` field

(REQUEST Mapping Template only)

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" : "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 Amazon ES 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

(REQUEST Mapping Template only)

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 (p. 207).

Resolver Mapping Template Reference for Lambda

This is prerelease documentation for a service in preview release. It is subject to change.

The AWS AppSync Lambda resolver mapping templates enable you to shape requests from AWS AppSync to AWS Lambda functions located in your account, and responses from your Lambda functions back to AWS AppSync. Mapping templates also enable you 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 AWS Lambda operations.

Request Mapping Template

The Lambda request mapping template is fairly simple and allows as much context information as possible to pass to your Lambda function.

```
{
   "version": string,
   "operation": Invoke|BatchInvoke,
   "payload": any type
}
```

Here is the JSON schema representation of the Lambda request mapping template, once resolved.

```
{
    "definitions": {},
    "$schema": "http://json-schema.org/draft-06/schema#",
    "$id": "http://aws.amazon.com/appsync/request-mapping-template.json",
    "type": "object",
    "properties": {
        "version": {
            "$id": "/properties/version",
            "type": "string",
            "enum": [
                "2017-02-28"
            ],
            "title": "The Mapping template version.",
            "default": "2017-02-28"
        },
        "operation": {
            "$id": "/properties/operation",
            "type": "string",
            "enum": [
                "Invoke",
                "BatchInvoke"
            ],
            "title": "The Mapping template operation.",
            "description": "What operation to execute.",
            "default": "Invoke"
        },
        "payload": {}
    },
    "required": [
        "version",
        "operation"
    ],
    "additionalProperties": false
}
```

Here is an example where we chose to pass the `field` value, and the GraphQL field arguments from the context.

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

The entire mapping document will be passed as input to your Lambda function, so that the previous example would now look like the following:

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": {
            "id": "postId1"
        }
    }
}
```

version

Common to all request mapping templates, `version` defines the version that the template uses. `version` is required.

```
"version": "2017-02-28"
```

operation

The Lambda data source allows you to define two operations, `Invoke` and `BatchInvoke`. The `Invoke` lets AWS AppSync know to call your Lambda function for every GraphQL field resolver, while `BatchInvoke` instructs AWS AppSync to batch requests for the current GraphQL field.

For **Invoke**, the resolved request mapping template exactly matches the input payload of the Lambda function. So the following sample template:

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "arguments": $utils.toJson($context.arguments)
    }
}
```

is resolved and passed to the Lambda function, as follows:

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "arguments": {
            "id": "postId1"
        }
    }
}
```

For **BatchInvoke**, the mapping template is applied for every field resolver in the batch. For conciseness, AWS AppSync merges all of 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": "2017-02-28",
    "operation": "BatchInvoke",
    "payload": $utils.toJson($context)
}
```

This template is resolved into the following mapping document:

```
{
    "version": "2017-02-28",
    "operation": "BatchInvoke",
    "payload": [
        {...}, // context for batch item 1
        {...}, // context for batch item 2
        {...}  // context for batch item 3
    ]
}
```

where 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, as follows:

```
[
    { "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
]
```

`operation` is required.

payload

The `payload` field is a container that you can use to pass any well-formed JSON to the Lambda function.

If the `operation` field is set to `BatchInvoke`, AWS AppSync will wrap the existing `payload` values into a list.

`payload` is optional.

Response Mapping Template

As with other data sources, your Lambda function sends a response to AWS AppSync that needs to be converted to a GraphQL type.

The result of the Lambda function will be set on the `context` object that is available via the 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:

```
$utils.toJson($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`, the Lambda function response must follow this structure:

```
[{
   "data": string,
   "errorMessage": string,
   "errorType": string
}]
```

If `data` is `null` and there is no error for the current item, the entire object can be replaced with null, for no data.

Also, if `errorMessage` is provided, `errorType` must also be provided, and vice versa.

Here is an example response that encompasses all of the cases we mentioned previously:

```
[
    { "data": "Author ABC data", "errorMessage": null, "errorType": null },
    { "data": "Author DEF data", "errorMessage": "Incomplete result", "errorType":
 "INCOMPLETE" },
    { "data": null, "errorMessage": "Failed to retrieve author", "errorType": "FAILED" },
    { "data": null, "errorMessage": null, "errorType": null }, // no data
    null  // no data
]
```

Here is the JSON schema representation of the Lambda function response:

```
{
    "definitions": {},
    "$schema": "http://json-schema.org/draft-06/schema#",
    "$id": "http://aws.amazon.com/appsync/lambda-response.json",
    "type": "array",
    "items": {
        "type": "object",
        "properties": {
            "data": {},
            "errorMessage": {
                "type": "string",
                "title": "Error message to be propagated to the response."
            },
            "errorType": {
                "type": "string",
                "title": "Error type to be propagated to the response."
            }
        },
        "dependencies": {
            "errorMessage": {
                "required": [
                    "errorType"
                ]
            },
            "errorType": {
                "required": [
                    "errorMessage"
                ]
            }
        }
    }
}
```

Resolver Mapping Template Reference for None Data Source

This is prerelease documentation for a service in preview release. It is subject to change.

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 a 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": {},
    "$schema": "http://json-schema.org/draft-06/schema#",
    "$id": "http://aws.amazon.com/appsync/request-mapping-template.json",
    "type": "object",
    "properties": {
        "version": {
            "$id": "/properties/version",
            "type": "string",
            "enum": [
                "2017-02-28"
            ],
            "title": "The Mapping template version.",
            "default": "2017-02-28"
        },
        "payload": {}
    },
    "required": [
        "version"
    ],
    "additionalProperties": false
}
```

Here is an example where we chose to pass the field arguments via the VTL context property `$context.arguments`:

```
{
    "version": "2017-02-28",
    "payload": $utils.toJson($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, `version` defines the version used by the template.

`version` is required.

Example:

```
"version": "2017-02-28"
```

payload

The `payload` field is a container that can be used to pass any well-formed JSON to the response mapping template.

`payload` 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:

```
$utils.toJson($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.

Resolver Mapping Template Context Reference

AWS AppSync defines a set of variables and functions for working with resolver mapping templates to make logicfull operations on data easier with GraphQL. This document describes those functions and provides examples for working with templates.

Accessing the `$context`

The `$context` variable holds all of the contextual information for your resolver invocation. It has the following structure:

```
{
   "arguments" : { ... },
   "source" : { ... },
   "result" : { ... },
   "identity" : { ... }
}
```

Each field is defined as follows:

`arguments`

A map containing all GraphQL arguments for this field.

`identity`

An object containing information about the caller. See Identity (p. 206) for more information on the structure of this field.

`source`

A map containing the resolution of the parent field.

`result`

A map containing the results of this resolver. This map is only available to response mapping templates.

For example, if you are 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" : {},
   "source" : {
      "createdAt" : "2017-02-28T18:12:37Z",
      "title" : "A new post",
      "content" : "A long time ago, in a thread far far away",
      "postId" : "1234",
      "authorId" : "34521"
   },
   "result" : {
      "name" : "Steve",
      "joinDate" : "2017-02-28T18:12:37Z",
      "id" : "34521"
   },
   "identity" : {
      "sourceIp" : "x.x.x.x",
      "userArn" : "arn:aws:iam::123456789012:user/appsync",
      "accountId" : "123456789012",
      "user" : "AIDAAAAAAAAAAAAAAAAAA"
   }
}
```

Identity

The `identity` section contains information about the caller. The shape of this section depends on the authorization type of your AWS AppSync API.

See Authorization Use Cases (p. 141) for more information about this section and how it can be used.

`API_KEY` authorization

The `identity` field is not populated.

`AWS_IAM` authorization

The `identity` has the following shape:

```
{
    "accountId" = "string",
    "cognitoIdentityPoolId" = "string",
    "cognitoIdentityId" = "string",
    "sourceIp" = "string",
    "username" = "string", // IAM user principal
    "userArn" = "string"
}
```

`AMAZON_COGNITO_USER_POOLS` authorization

The `identity` has the following shape:

```
{
    "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 the user has.

`cognitoIdentityId`

The Amazon Cognito identity ID of the caller.

`cognitoIdentityPoolId`

The Amazon Cognito identity pool ID associated with the caller.

`defaultAuthStrategy`

The default auth strategy for this caller (`ALLOW` or `DENY`).

`issuer`

The token issuer.

`sourceIP`

The source IP address of the caller.

`sub`

The UUID of the authenticated user.

`user`

The IAM user.

`userArn`

The IAM user ARN.

`username`

The username of the authenticated user. In case of `AMAZON_COGNITO_USER_POOLS` authorization, the value of username is the value of attribute *cognito:username*. In case of `AWS_IAM` authorization, the value of the username is the value of the AWS User Principal. We recommend that you use `cognitoIdentityId` if you are using *AWS IAM* authorization with credentials vended from Amazon Cognito Federeated Identities.

Access Request Headers

AWS AppSync supports passing custom headers from clients and accessing them in your GraphQL resolvers using `$context.request.headers`. You can then use the header values for actions like inserting data to a data source or even authorization checks. Single or multiple request headers can use used as shown in the following examples using `$curl` with an API key from the command line:

Single Header Example

Suppose you set a header of `custom` with a value of `nadia` like so:

```
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": { "S": "$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 was 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.random[1]"`.

Utility Helpers in $util

The `$util` variable contains general utility methods that make it easier to work with data.

Unless otherwise specified, all utilities use the UTF-8 character set.

`$util.escapeJavaScript(String) : String`

Returns the input string as a JavasScript 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.unauthorized()`

Throws `Unauthorized` for the field being resolved. This can be used in request or response mapping templates to decide if the caller should be allowed to resolve the field.

`$util.error(String)`

Throws a custom error. This can be used in request or response mapping templates if the template detects an error with the request.

`$util.error(String, String)`

Throws a custom error. This can be used in request or response mapping templates if the template detects an error with the request. Additionally, an `errorType` can be specified.

`$util.error(String, String, Object)`

Throws a custom error. This can be used in request or response mapping templates if the template detects an error with the request. Additionally, an `errorType` and a `data` 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.

`$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.

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. Below we provide some examples, as well as a list of available methods and descriptions.

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

```
$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
```

`$util.time.nowISO8601() : String`

Returns a String represetation 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, and return 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, and return the timestamp as milliseconds since epoch.

`$util.time.parseISO8601ToEpochMilliSeconds(String) : Long`

Parses an ISO8601 timestamp, passed as a String, and return 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 a epoch milliseconds timestamp to an ISO8601 timestamp.

`$util.time.epochMilliSecondsToFormatted(long, String) : String`

Converts a 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 a epoch milliseconds timestamp, passed as a long, to a timestamp formatted according to the supplied format in the supplied timezone.

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.

`$util.list.copyAndRetainAll(List, List) : List`

Makes a shallow copy of the supplied list in the first argument, 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, removing any items where the item is specified in the second arguement, if they are present. All other items will be retained in the copy.

Map Helpers in $util.map

`$util.map` contains methods to help with common List operations, such as removing or retaining items from a list for filtering use cases.

`$util.map.copyAndRetainAllKeys(Map, Map) : Map`

Makes a shallow copy of the first map, retaining only the keys specified in the second map, if they are present. All other keys will be removed from the copy.

`$util.map.copyAndRemoveAllKeys(Map, Map) : Map`

Makes a shallow copy of the first map, removing any entries where the key is specified in the second map, if they are present. All other keys will be retained in the copy.

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. 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" : "$util.autoId()} }
    },
    "attributeValues" : {
         "title" : { "S" : "${ctx.args.title}" },
         "author" : { "S" : "${ctx.args.author}" },
         "version" : { "N", $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 version: a version that returns an object (eg., `$util.dynamodb.toString(...)`), and a version that returns the object as a JSON string (e.g., `$util.dynamodb.toStringJson(...)`). In the previous example, we used the version that returns the data as a JSON string. If you wanted to manipulate the object before being used in the template, you can choose to return an object instead:

```
{
    "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.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!"))
```

`$util.dynamodb.toDynamoDB(Object) : Map` `$util.dynamodb.toDynamoDBJson(Object) : String`

General object conversion tool for DynamoDB. Converts input Objects to the appropriate DynamoDB format. Opinionated about type inference: e.g., use lists ("L") rather than sets ("SS", "NS", "BS").

String example:

```
Input:      $util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

Number example:

```
Input:      $util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

Note: This is actually a number, not a string representing a number.

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" }
                       ]
                   }
               }
           }
```

`$util.dynamodb.toString(String) : Map` `$util.dynamodb.toStringJson(String) : String`

Converts input Strings to the appropriate DynamoDB String format.

```
Input:      $util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

`$util.dynamodb.toStringSet(List<String>) : Map` `$util.dynamodb.toStringSetJson(List<String>) : String` Converts Lists with Strings to the appropriate DynamoDB List format.

```
Input:      $util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

`$util.dynamodb.toNumber(Number) : Map` `$util.dynamodb.toNumberJson(Number) : String` Converts numbers to the appropriate DynamoDB Number format.

```
Input:      $util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

Note: This is actually a number, not a string representing a number.

`$util.dynamodb.toNumberSet(List<Number>) : Map` `$util.dynamodb.toNumberSetJson(List<Number>) : String` Converts List of numbers to the appropriate DynamoDB List format.

```
Input:      $util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

Note: This is actually a number, not a string representing a number.

`$util.dynamodb.toBinary(String) : Map` `$util.dynamodb.toBinaryJson(String) : String` Converts Strings to the appropriate DynamoDB format.

```
Input:      $util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

Note: Binary data should be represented as a base64-encoded string.

`$util.dynamodb.toBinarySet(List<String>) : Map`
`$util.dynamodb.toBinarySetJson(List<String>) : String`

```
Input:      $util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

Note: Binary data should be represented as a base64-encoded string.

`$util.dynamodb.toBoolean(boolean) : Map` `$util.dynamodb.toBooleanJson(boolean) : String`

```
Input:      $util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

`$util.dynamodb.toNull() : Map` `$util.dynamodb.toNullJson() : String`

```
Input:      $util.dynamodb.toNull()
Output:     { "NULL" : null }
```

`$util.dynamodb.toList(List) : Map` `$util.dynamodb.toListJson(List) : String`

```
Input:      $util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
            }
```

`$util.dynamodb.toMap(Map) : Map` `$util.dynamodb.toMapJson(Map) : String`

```
Input:      $util.dynamodb.toMap({ "foo": "bar", "baz" : 1234, "beep": [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
                           { "S" : "boop" }
                       ]
                   }
               }
            }
```

`$util.dynamodb.toMapValues(Map) : Map` `$util.dynamodb.toMapValuesJson(Map) : String`

```
Input:      $util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep": [ "boop"] })
Output:     {
               "foo"  : { "S" : "bar" },
               "baz"  : { "N" : 1234 },
               "beep" : {
                   "L" : [
                       { "S" : "boop" }
                   ]
               }
           }
```

`$util.dynamodb.toS3Object(String key, String bucket, String region) : Map` `$util.dynamodb.toS3ObjectJson(String key, String bucket, String region) : String`

```
Input:      $util.dynamodb.toS3Object("foo", "bar", region = "baz")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo", \"bucket\" : \"bar", \"region\" :
 \"baz" } }" }
```

`$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map` `$util.dynamodb.toS3ObjectJson(String key, String bucket, String region, String version : String`

```
Input:      $util.dynamodb.toS3Object("foo", "bar", "baz", "beep")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region\" :
 \"baz\", \"version\" = \"beep\" } }" }
```

`$util.dynamodb.fromS3ObjectJson(String) : Map`

```
Input:      $util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
 \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" : "beep" }
```

Troubleshooting and Common Mistakes

This is prerelease documentation for a service in preview release. It is subject to change.

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" : { "S" : "${context.arguments.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.

Integrating with Amazon CloudWatch

This is prerelease documentation for a service in preview release. It is subject to change.

When you interact with AWS AppSync, it sends the following metrics and dimensions to Amazon CloudWatch every minute. You can use the following procedures to view the metrics for AWS AppSync.

You can monitor AWS AppSync using CloudWatch, which collects and processes raw data from AWS AppSync into readable, near real-time metrics. These statistics are recorded for a period of two weeks, so that you can access historical information and gain a better perspective on how your web application or service is performing. By default, AWS AppSync metric data is sent to CloudWatch in one-minute intervals. For more information, see What Is Amazon CloudWatch in the *Amazon CloudWatch User Guide*.

You must have the appropriate CloudWatch permissions to monitor AWS AppSync with CloudWatch. For more information, see Authentication and Access Control for Amazon CloudWatch in the *Amazon CloudWatch User Guide*.

Getting CloudWatch Metrics (CLI)

The following code displays available metrics for AWS AppSync.

```
aws cloudwatch list-metrics --namespace "AWS/AppSync"
```

AWS AppSync Metrics

AWS AppSync produces the following metrics for each request. These metrics are aggregated and sent to CloudWatch.

Metric	Description
4xxError	The number of requests that result in client-side error. Unit: Count
Throttles	The number of requests that are throttled. Unit: Count
Count	The number of calls to the API. Unit: Count

Metric	Description
Latency	The time between when AWS AppSync receives a request from a client and when it returns a response to the client. Unit: Millisecond

AWS AppSync Dimensions

AWS AppSync produces the following dimensions for each request.

Dimension	Description
GraphQLAPIId	Filters AWS AppSync metrics for an API with the specific ID.
Operation	Filters AWS AppSync metrics for an API with the specific operation (query, mutation, or subscription).
GraphQLAPIId, Operation	Filters AWS AppSync metrics for an API with the specific ID and operation.

Logging AWS AppSync API Calls with AWS CloudTrail

This is prerelease documentation for a service in preview release. It is subject to change.

AWS AppSync is integrated with AWS CloudTrail, a service that captures API calls made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. CloudTrail captures API calls from the AWS AppSync console or from the AWS AppSync API. Using the information collected by CloudTrail, you can determine what request was made to AWS AppSync, the source IP address from which the request was made, who made the request, when it was made, and so on. To learn more about CloudTrail, including how to configure and enable it, see What Is Amazon CloudTrail in the *Amazon CloudTrail User Guide*.

AWS AppSync Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, API calls made to AWS AppSync actions are tracked in CloudTrail log files. AWS AppSync records are written together with other AWS service records in a log file. CloudTrail determines when to create and write to a new file based on a time period and file size.

All AWS AppSync actions are logged by CloudTrail and are documented in the AWS AppSync API Reference. For example, calls to the `CreateGraphqlApi`, `CreateDataSource`, and `ListResolvers` actions generate entries in the CloudTrail log files.

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 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.

You can also create a trail and store your log files in your Amazon S3 bucket for as long as you want, and define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted with Amazon S3 server-side encryption (SSE).

To be notified of log file delivery, configure CloudTrail to publish Amazon SNS notifications when new log files are delivered. For more information, see Configuring Amazon SNS Notifications for CloudTrail.

You can also aggregate AWS AppSync log files from multiple AWS regions and multiple AWS accounts into a single Amazon S3 bucket.

For more information, see Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts.

Understanding AWS AppSync Log File Entries

CloudTrail log files can contain one or more log entries. Each entry lists multiple JSON-formatted events. A log entry 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. Log entries are not an ordered stack trace of the public API calls, so they do not appear in any specific order.

Because of potential confidentiality issues, log entries do not contain the synthesized text. Instead, this text is redacted in the log entry.

The following example shows a CloudTrail log entry that demonstrates the `CreateApiKey` action.

```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::123456789012:user/Alice",
      "accountId": "123456789012",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "Alice"
    },
    "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": "123456789012"
    }
  ]
}
```

The following example shows a CloudTrail log entry that demonstrates the `ListApiKeys` action.

```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::123456789012:user/Alice",
      "accountId": "123456789012",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "Alice"
    },
    "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": "123456789012"
    }
  ]
}
```

The following example shows a CloudTrail log entry that demonstrates the `DeleteApiKey` action.

```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::123456789012:user/Alice",
      "accountId": "123456789012",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "Alice"
    },
    "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": "123456789012"
    }
  ]
}
```