(Optional) Deep Dive on AWS AppSync

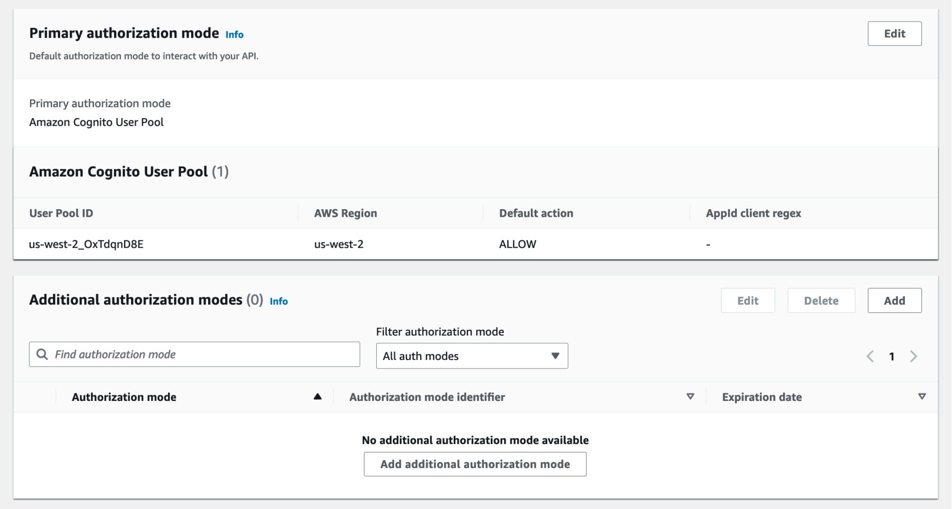
Since the Chatbot Playground API was preconfigured, this section serves to explain how the AppSync API was built. This section will cover

- Authorization Modes
- GraphQL Schema
- Unit Resolvers
- Environment Variables
- HTTP Data Sources

Authorization Modes

Whenever you are making a user-facing API, you must consider how you will protect your API. While Web Application Firewall and Private APIs are out of scope of this workshop, you may have noticed a requirement to authenticate into a User Pool with credentials that were created for you in this workshop. Let's see where this is configured.

1. Head to the AWS AppSync console
2. Navigate to **Settings** in the left navigation menu
3. Scroll to the section titled **Primary authorization mode**
4. Note the configuration for the **Amazon Cognito User Pool**



Primary authorization mode

Default authorization mode to interact with your API.

Primary authorization mode
Amazon Cognito User Pool

Amazon Cognito User Pool (1)

User Pool ID	AWS Region	Default action	AppId client regex
us-west-2_OxTdqnD8E	us-west-2	ALLOW	-

Additional authorization modes (0)

Filter authorization mode
All auth modes

Authorization mode	Authorization mode identifier	Expiration date

No additional authorization mode available

This workshop came with a preconfigured Amazon Cognito User Pool, a service that provides a directory of users for use in Customer Identity and Access Management (CIAM). The authentication mode configured here required all API client consumers to provide Cognito User Pool Tokens (Open ID Connect JavaScript Web Tokens) in order to access any API Operation.

If you wanted to add additional Authorization Modes, such as AWS IAM, API Key, OIDC, or Lambda, you could select which Authorization Mode is approved for which API operation to provide fine-grained access control. This is useful if you need one of your backend services to use IAM credentials when acting as a GraphQL API Client to perform privileged operations that external users should not have access to.

GraphQL Schema

GraphQL is an API Protocol that organizes API Operations into groups of Types and Fields. If you are familiar with Graphs, the Types in your GraphQL Schema can be thought of as nodes, and Fields on that Type are like the edges/relationships on a Graph. The structure and purpose of a

GraphQL API is ecapsulated in its Schema and it defines the client access patterns and payload requirements.

GraphQL APIs offer introspection so that clients may see the operations available on an API and what data they must provide for certain operations. API Clients may also specify which fields they would like returned, which reduces the over- and under-fetching challenges that arise in RESTful API designs.

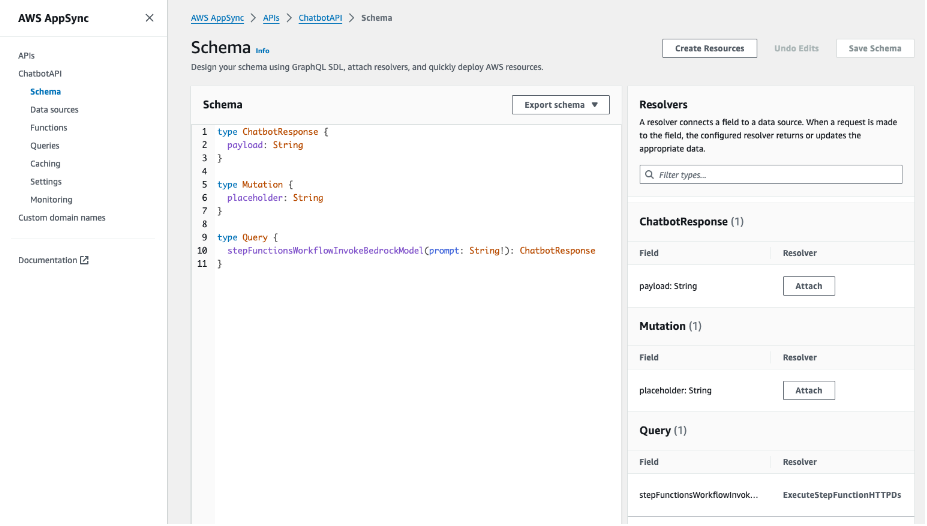
GraphQL APIs are useful for both simple APIs and complex API topologies, where your API provides access to data models with nested relationships from multiple data sources.

Let's look at the GraphQL Schema in the AppSync Console

1. Navigate to **Schema** in the left navigation menu
2. Look at the two root types **Query** and **Mutation**

- Query fields are intended as operations that do not mutate data sources. These may include **Read** and **List** operations on a Database
- Mutation fields are intended as operations that do mutate data sources. These operations may include **Create**, **Update**, and **Delete** operations. Because the StepFunction Execution does not mutate any data, the `stepFunctionsWorkflowInvokeBedrockModel` field was placed on the `Query` root type.
- Subscription fields are ways for you to subscribe to realtime updates on your API. When mutations are triggered, clients that have a websocket connection to the subscription endpoint will stream the results of the mutation operation

3. In the GraphQL Schema page of the AppSync Console, you will see the Resolver section. A resolver connects a field to a data source. This is how we created the connection to the **ChatbotPlaygroundWorkflow** Express State Machine from AppSync.



Design your schema using GraphQL SDL, attach resolvers, and quickly deploy AWS resources.

```
type ChatbotResponse {
  payload: String
}

type Mutation {
  placeholder: String
}

type Query {
  stepFunctionsWorkflowInvokeBedrockModel(prompt: String!): ChatbotResponse
}
```

A resolver connects a field to a data source. When a request is made to the field, the configured resolver returns or updates the appropriate data.

ChatbotResponse (1)

Field	Resolver
payload: String	Attach

Mutation (1)

Field	Resolver
placeholder: String	Attach

Query (1)

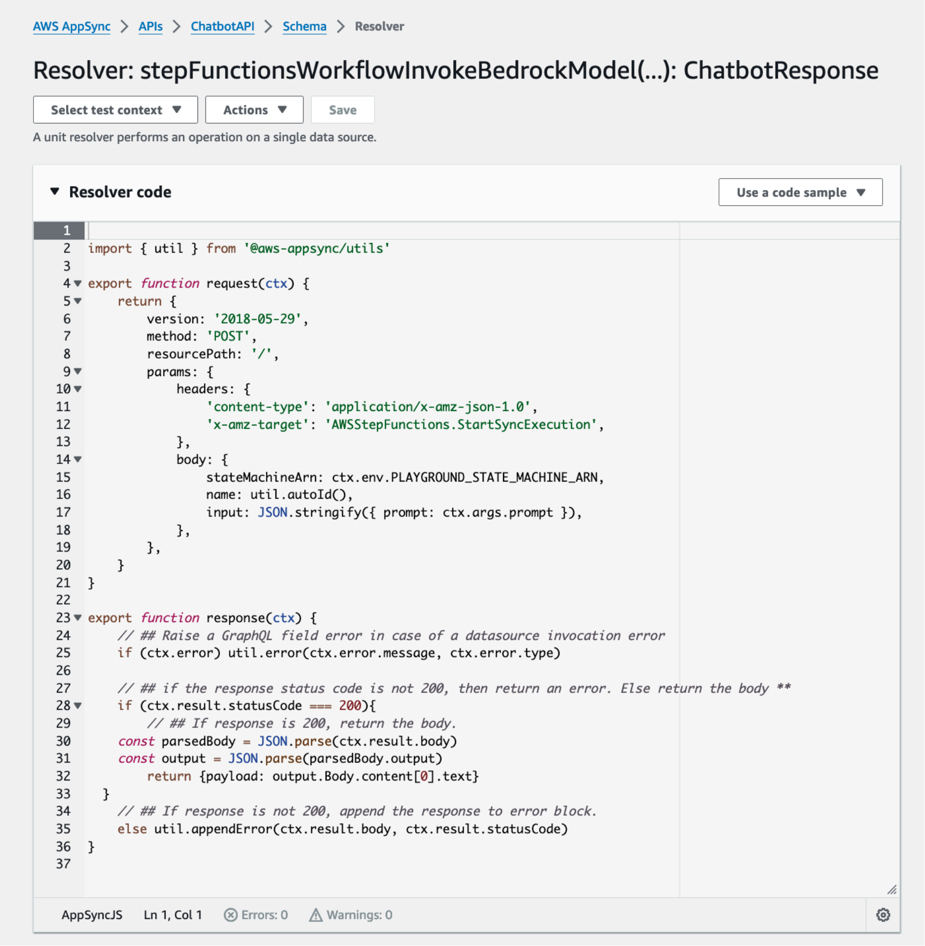
Field	Resolver
stepFunctionsWorkflowInvok...	ExecuteStepFunctionHTTPDs

Unit Resolver and HTTP Data Source

1. Select the **ExecuteStepFunctionHTTPDs** hyperlink under the Resolvers window to the right of the GraphQL Schema. This item is written in bold font at the bottom right of the above screenshot.
2. Looking at the Resolver Code, you will see some business logic in this HTTP resolver. With AWS AppSync, you can connect to any AWS Service by using a normal HTTP Resolver. Here you will see two functions: `request` and `response`

- Request: This function executes immediately when the client's operation is received. In the code, you will notice that the resolver returns a **POST** request to perform a **StartSyncExecution** on your stateMachineArn defined by your environment variable **PLAYGROUND_STATE_MACHINE_ARN**. This is also where the prompt provided in the GraphQL Query is provided as input to the State Machine Execution.
- Response: This function executes after the Data Source responds to the request. In your Express Workflow synchronous execution response, you will see the Resolver Code parses the payload

from the output's body's contents.


AWS AppSync > APIs > ChatbotAPI > Schema > Resolver
Resolver: stepFunctionsWorkflowInvokeBedrockModel(...): ChatbotResponse
Select test context ▼
Actions ▼
Save
A unit resolver performs an operation on a single data source.
▼ Resolver code
Use a code sample ▼
AppSyncJS
Ln 1, Col 1
Errors: 0
Warnings: 0

```

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

export function request(ctx) {
    return {
        version: '2018-05-29',
        method: 'POST',
        resourcePath: '/',
        params: {
            headers: {
                'content-type': 'application/x-amz-json-1.0',
                'x-amz-target': 'AWSStepFunctions.StartSyncExecution',
            },
            body: {
                stateMachineArn: ctx.env.PLAYGROUND_STATE_MACHINE_ARN,
                name: util.autoId(),
                input: JSON.stringify({ prompt: ctx.args.prompt }),
            },
        },
    }
}

export function response(ctx) {
    // ## Raise a GraphQL field error in case of a datasource invocation error
    if (ctx.error) util.error(ctx.error.message, ctx.error.type)

    // ## if the response status code is not 200, then return an error. Else return the body **
    if (ctx.result.statusCode === 200){
        // ## If response is 200, return the body.
    const parsedBody = JSON.parse(ctx.result.body)
    const output = JSON.parse(parsedBody.output)
        return {payload: output.Body.content[0].text}
  }
    // ## If response is not 200, append the response to error block.
    else util.appendError(ctx.result.body, ctx.result.statusCode)
}

```

3. Expand the `Data Source` section and note how this resolver is tied to the ExecuteStepFunctionHTTPDs

Environment Variables

Environment Variables are key-value pairs that you configure at the API level. You may use Environment Variables to provide resource names/identifiers within your API Resolvers. You can also use the environment variables as values that you provide to your API Data Sources (such as a Lambda Function event argument) or as flags within your Business Logic in your AppSync Resolvers.

Note the lines in the below screenshot that access the environment variable via `ctx.env`

```
...
export function request(ctx) {
...
  body: {
    stateMachineArn: ctx.env.PLAYGROUND_STATE_MACHINE_ARN,
  ...
  }
}
...
```

HTTP Data Source

- Note how this Data Source has an HTTP Data Source Type on the `https://sync-states.<region>.amazonaws.com/` endpoint and is configured with an IAM Role. This IAM Role is what is used to sign the HTTP request from the GraphQL API to ensure the API has the permissions to

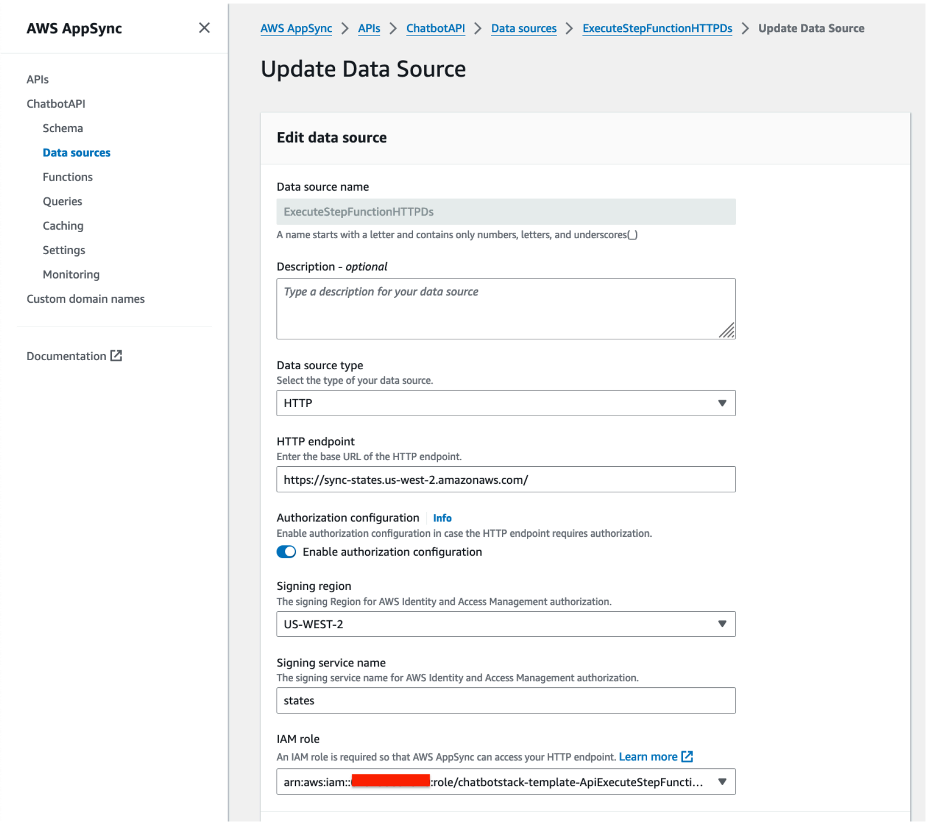
▼ AWS account access

Open AWS console (us-west-2)

Get AWS CLI credentials

Exit event

Start a Step Function Execution.



Update Data Source

Edit data source

Data source name
ExecuteStepFunctionHTTPDs
A name starts with a letter and contains only numbers, letters, and underscores(_)

Description - *optional*
Type a description for your data source

Data source type
Select the type of your data source.
HTTP

HTTP endpoint
Enter the base URL of the HTTP endpoint.
https://sync-states.us-west-2.amazonaws.com/

Authorization configuration Info
Enable authorization configuration in case the HTTP endpoint requires authorization.
● Enable authorization configuration

Signing region
The signing Region for AWS Identity and Access Management authorization.
US-WEST-2

Signing service name
The signing service name for AWS Identity and Access Management authorization.
states

IAM role
An IAM role is required so that AWS AppSync can access your HTTP endpoint. Learn more
arn:aws:iam::[redacted]:role/chatbotstack-template-ApiExecuteStepFuncti...

ⓘ Good job investigating the inner workings of your Chatbot API. Now you will have a better understanding of the backend integration when you consume this API from the client application

Previous Next