AWS Amplify Introduction

Sirvan Almasi

AWS Amplify[1] is a framework to rapidly create web and mobile applications (including APIs). It uses the rich AWS ecosystem to speed up development and deployment.

AWS Amplify includes an easy to use CLI, set of libraries for popular front-end libraries (e.g. react and vuejs), and UI components. Using AWS Amplify one can create **authentication**, **storage**, **API** (GraphQL and REST), **DataStore**, and much more.

[1] Amplify Framework Docs, September 2021

Authentication

API GraphQL

GraphQL[2] is a query language for your API. It is often compared to the REST architecture. We won't compare them here, but a simple Google search will provide you plenty of answers.

[2] graphql.org

- Assuming you have done `amplify init`, you can begin to add an API by doing `amplify add api`. Following the instructions will hopefully get you to where you need to go.
- You will be able to update your API schema at:

  `/amplify/backend/api/gqls3/schema.graphql`.

- `amplify push` will deploy your API to the AWS cloud services.
- Object types that are annotated with `@model` are top-level entities in the generated API. Objects annotated with `@model` are stored in Amazon DynamoDB and are capable of being protected via `@auth`, related to other objects via `@connection`, and streamed into Amazon OpenSearch via `@searchable`. You may also apply the `@versioned` directive to instantly add a version field and conflict detection to a model type.
- `createdOn` and `updatedOn` fields are automatically created for your models.

Amplify GraphQL Data Types:

- ID
- String
- Int
- Float
- Boolean
- AWSDate
- AWSTime
- AWSDateTime
- AWSTimeStamp
- AWSEmail
- AWSJSON
- AWSPhone
- AWSURL
- AWSIPAddress

Relationships

`@connection` enables you to specify relationships between models. 1-to-1, 1-to-many, and m-to-m are supported.

One-to-one

```
type Project @model {
  id: ID!
  name: String
  teamID: ID!
  team: Team @connection(fields: ["teamID"])
}

type Team @model {
  id: ID!
  name: String!
}
```

Figure 1: `Project` has one `Team` and we have `teamID` as an identifier for the team that the project belongs to.

One-to-many

This needs a `@key` item.

```
type Post @model {
  id: ID!
  title: String!
  comments: [Comment] @connection(keyName: "byPost", fields: ["id"])
}

type Comment @model
  @key(name: "byPost", fields: ["postID", "content"]) {
  id: ID!
  postID: ID!
  content: String!
}
```

Figure 2: `Post` can have many `Comment` s.

Belongs to

```
type Post @model {
  id: ID!
  title: String!
  comments: [Comment] @connection(keyName: "byPost", fields: ["id"])
}

type Comment @model
  @key(name: "byPost", fields: ["postID", "content"]) {
  id: ID!
  postID: ID!
  content: String!
  post: Post @connection(fields: ["postID"])
}
```

Figure 3: You can make a connection bi-directional by adding a many-to-one connection to types that already have a one-to-many connection. In this case you add a connection from Comment to Post since each comment belongs to a post:

Authorization

GraphQL Operation Types:

- `CREATE`
- `DELETE`
- `GET`
- `LIST`
- `UPDATE`

Models annotated with `@auth` are protected by a set of authorization rules. When using the `@auth` directive on object type definitions that are also annotated with `@model`, all resolvers that return objects of that type will be protected. When using the `@auth` directive on a field definition, a resolver will be added to the field that authorize access based on attributes found in the parent type.

We can protect our models (think of them as database tables) at **model level**; this means that all the fields in that model will have the same protection. We can also protect individual fields at **field level authorisation**. We can use both methods at the same time. This level of protection is required as we demand different CRUD access from our users.

We authorise access with respect to users. We can abstract our users to the following types:

- `owner`. The individual that *owns* the record.
- `groups`. Static groups of users that have shared permission, e.g. admins.
- `public`. Anyone can access the API.
- `private`. The private authorization specifies that everyone will be allowed to access the API with a valid JWT token from the configured Cognito User Pool.

```
type Post @model
  @auth (
    rules: [
      # allow all authenticated users ability to create posts
      # allow owners ability to update and delete their posts
      { allow: owner },

      # allow all authenticated users to read posts
      { allow: private, operations: [read] },

      # allow all guest users (not authenticated) to read posts
      { allow: public, operations: [read] }
    ]
  ) {
  id: ID!
  title: String
  owner: String
}
```

Model Level Authorization

```
type Post @model
  @auth(rules: [{ allow: owner }]) {
  id: ID!
  title: String!
}
```

Figure 4: In this schema, only the owner of the object has the authorization to perform read (getTodo and listTodos), update (updateTodo), and delete (deleteTodo) operations on the owner created object. This prevents the object from being updated or deleted by users other than the creator of the object.

```
type Todo @model
  @auth(rules: [{ allow: owner, operations: [create, delete, update] }]) {
  id: ID!
  updatedAt: AWSDateTime!
  content: String!
}
```

Figure 5: This `@auth` rule says: only the owner can delete, and update this model. Others can therefore read (get and list) and create.

Field Level Authorization

- You may dictate field level authorisation. E.g. you may only want an admin to `update` a certain field.

```
type Employee @model {
  id: ID!
  email: String
  username: String

  # Owners & members of the "Admin" group may read employee salaries.
  # Only members of the "Admin" group may create an employee with a salary
  # or update a salary.
  salary: String
    @auth(rules: [
      { allow: owner, ownerField: "username", operations: [read] },
      { allow: groups, groups: ["Admin"], operations: [create, update, read]
      }
    ])
}
```

```
type User @model {
  id: ID!
  username: String
  ssn: String @auth(rules: [
      { allow: owner,
      ownerField: "username" }
    ])
}
```

You might want to have a user model where some fields, like username, are a part of the public profile and the ssn field is visible to owners.

References

[1] Amplify Framework Docs, September 2021.