Collaborative Data Driven Web Applications with Amazon AppSync

Dimitra Zuccarelli, 20072495

December 2018

Contents

1 Introduction

The aim of this project is to explore and practically deploy several different products provided by Amazon Web Services. More specifically, the project will focus on two of AWS' most recent services, Amazon AppSync and AWS Amplify, which are designed to assist developers in the creation of mobile and web based cloud native applications. Other services that will be used include: AWS Lambda, AWS Cognito, AWS IAM, API Gateway and Amazon Comprehend.

A Javascript web application called `Jukebox` will be developed in order to highlight and demonstrate the features of AppSync and Amplify. The application will contain the following functionality:

- Allow users to sign up or log in to the application using Cognito User Pools
- Allow users to create playlists/jukeboxes
- Allow users to join a jukebox by its ID
- Add songs to the jukebox in real time. Any songs that are added to a jukebox should be synchronised to each instance of the `Jukebox` application immediately
- Any songs that are added to the jukebox when the user is offline should be uploaded when the user comes back online
- A jukebox's mood will be detected and displayed to the user using Amazon Comprehend's Natural Language Processing (NLP).

2 Technology Overview

2.1 GraphQL

AppSync is based on GraphQL, an open API standard which is often used to overcome certain limitations of traditional RESTful APIs. GraphQL is based on the idea of replacing multiple "dumb" endpoints with a single "smart" endpoint, which is capable of taking in complex queries and returning only what the client requires [4]. One powerful advantage of GraphQL is the ability to fetch all the data needed for a certain request in a single round-trip. The GraphQL server acts as an interface between the application and (potentially) multiple different data stores, which also means that legacy systems as well as new systems can be unified behind a single, coherent API [3, 5].

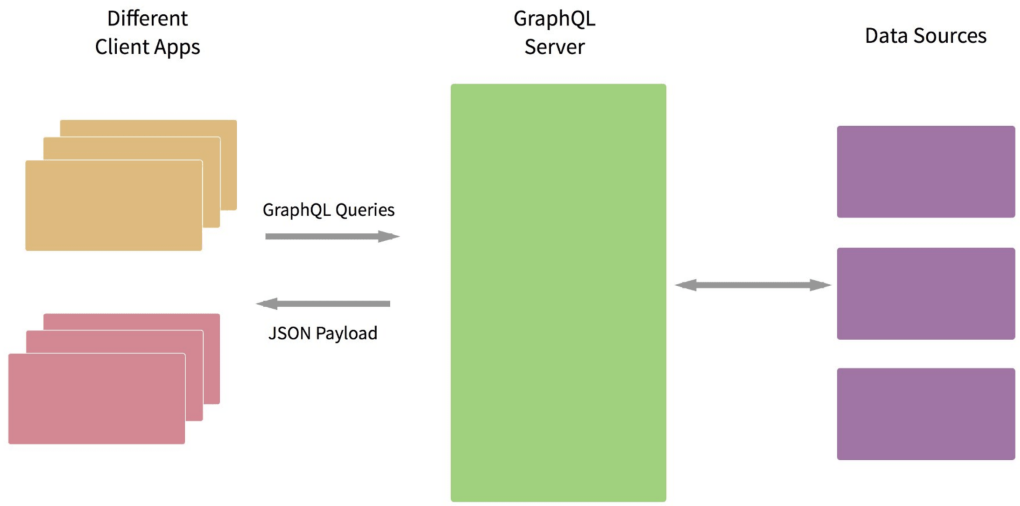


Figure 1: GraphQL server (Scotch.io, 2017)

The four main components of GraphQL are `queries`, `mutations`, `resolvers` and `subscriptions`. These will be described in detail below.

2.1.1 Queries

Queries are simply requests for data made to the GraphQL server. This would correspond to a `GET` request in traditional REST APIs. To illustrate the use of queries, Github's GraphQL API will be used.

An example of a query would be to request the username (`login`) and location of the currently logged in user (`viewer`) from Github's GraphQL endpoint. Note that the query is prefixed by the keyword `query`:

```
query {
  viewer {
    login
    location
  }
}
```

The `viewer` object could contain many different fields that are not necessarily of interest for the task at hand. One powerful feature of GraphQL is the ability to request (and receive) only exactly what is needed. In this case, only the username and the location were returned, as expected:

```
{
  "data": {
    "viewer": {
      "login": "dimitraz",
      "location": "Ireland"
    }
  }
}
```

A query can also specify arguments as filters. The next example contains a query for information about the first pull request on the `jukebox` repo. The requested fields are the time the pull request was merged (`mergedAt`) and the title of the pull request:

```
query {
  repository(name: "jukebox", owner: "dimitraz") {
    name
    pullRequest(number: 1) {
      title
      mergedAt
    }
  }
}
```

The result of the query looks like this:

```
{
  "data": {
    "repository": {
      "name": "jukebox",
      "pullRequest": {
        "title": "Configure Amplify, GraphQL",
        "mergedAt": "2018-10-08T10:11:48Z"
      }
    }
  }
```

```
}
```

Finally, queries also support variables for dynamic argument values. Using variables means arguments can be dynamically updated without ever having to change the query itself. The following example gets the first `n` amount of followers for the user `satyanadella`, and returns the name of each follower.

```
query($number_of_followers: Int!) {
  user(login: "satyanadella") {
    name
    location
    bio
    followers(first: $number_of_followers) {
       nodes {
        name
      }
    }
  }
}

variables {
  "number_of_followers": 5
}
```

Note that the variables section is prefixed by the `variables` keyword.
The result of the query looks like this:

```
{
  "data": {
    "user": {
      "name": "Satya Nadella",
      "location": "Redmond, Washington",
      "bio": "CEO of Microsoft",
      "followers": {
        "nodes": [
          { "name": "Bing" },
          { "name": "Yasuaki Matsuda" },
          { "name": "Manos Stamatakis" },
          { "name": "Tanner Harper" },
          { "name": "iqbalhossen1993" }
        ]
      }
    }
  }
}
```

2.1.2 Mutations

Mutations are used to modify data on the server side. In REST this would be accomplished using `POST`, `PUT` and `DELETE`, to create, update or delete objects.

An example of a mutation using the Github GraphQL API would be to add a reaction to an issue or a comment. The following example shows how reacting with a thumbs up emoji to an issue would be accomplished using the API.

Firstly, find the ID of the issue to add a reaction to:

```
query findIssueId {
  repository(name: "jukebox", owner: "dimitraz") {
    issue(number: 4) {
      id
```

```
    }
  }
}
```

Note that a query can be named, e.g. `"findIssueId"`.

Next, the `addReaction` mutation must be invoked, which requires an input of type `AddReactionInput`. This input is defined in the `reaction` variable, which contains the ID of the issue to react to and the emoji to react with.

```
mutation addIssueReaction($reaction: AddReactionInput!) {
  addReaction(input: $reaction) {
    reaction {
      content
      user {
        login
      }
    }
    subject {
      id
    }
  }
}

variables {
  "reaction": {
    "subjectId": "MDU6SXNzdWUzNzY1MDMyMzk=",
    "content": "THUMBS_UP"
  }
}
```

Note that with mutations, as with queries, the return output fields must be defined. Here the requested output is the content of the reaction (which should be the thumbs up emoji), the username of the user who added the reaction, and the ID of the issue that the reaction was added to.

The result of the mutation:

```
{
  "data": {
    "addReaction": {
      "reaction": {
        "content": "THUMBS_UP",
        "user": {
          "login": "dimitraz"
        }
      },
      "subject": {
        "id": "MDU6SXNzdWUzNzY1MDMyMzk="
      }
    }
  }
}
```

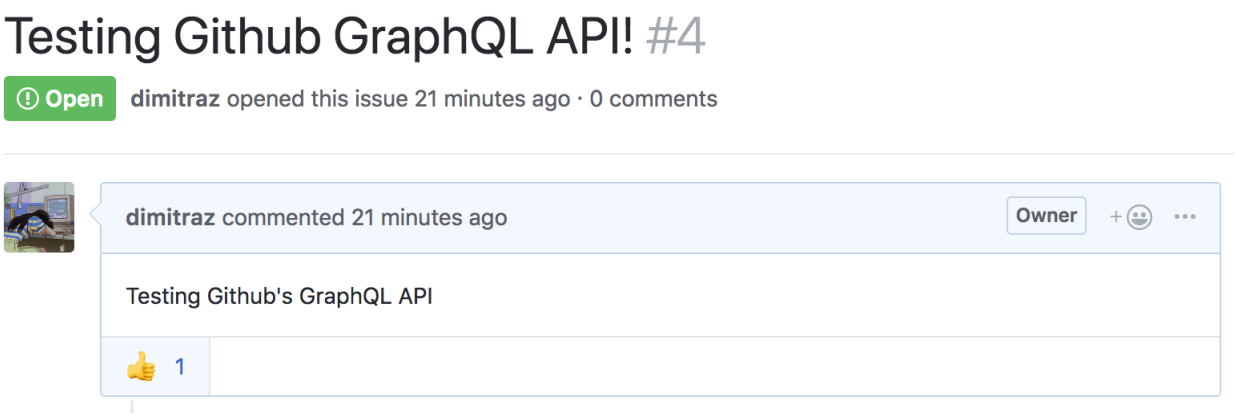


Figure 2: Adding a thumbs up reaction to an issue

2.1.3 Resolvers

Resolvers are the instructions that transform GraphQL operations into actual data. This could mean providing the database connection and request string for fetching a list of users, for example. In a traditional MEAN app, this might be done using Mongoose for access to the MongoDB database:

Listing 1: The User schema

```
// models/user.js
import mongoose from 'mongoose';

const Schema = mongoose.Schema;
const UserSchema = new Schema({
  username: String,
  email: String,
  bio: String,
}, {timestamps: true});

export default mongoose.model('User', UserSchema);
```

Next, when the `/api/users` endpoint is hit, the list of users is fetched from the database:

```
// routes/routes.js
import User from '../models/user';

// Get list of users
router.get('/users', async (req, res) => {
    let users;
    users = await User.find();
});
```

In GraphQL the process is very similar, with the main difference being that instead of mapping an endpoint to an action, the name of the query is mapped to that action. The list of resolvers must also be wrapped in a `resolvers` object.

The resolver for fetching all users from the database for the simple user would look like:

```
// resolvers.js
import User from './models/user';

export const resolvers = {
    Query: {
        async allUsers() {
```

```
            return await User.find();
        }
    }
};
```

One advantage of using AppSync is that it abstracts the complexity of creating all of the necessary GraphQL resolvers. This will be discussed later in section 2.2.

2.1.4 Subscriptions

Subscriptions are a feature of GraphQL which allow the server to send data to the client when a specific event (such as the creation or deletion of an object) happens. This is typically implemented using web sockets, allowing the server and the client to maintain a constant connection. This means that clients can react to events in real time, giving the user immediate feedback and allowing developers to create collaborative applications.

An example of a subscription in AWS AppSync looks like this:

```
type Subscription {
  onCreateUser: User
    @aws_subscribe(mutations: ["createUser"])
}
```

Subscriptions in AWS AppSync are invoked as a response to a mutation, which is why the `@aws_subscribe` directive is needed. This is all that is needed to get real time notifications from AppSync - when a client subscribes to this query, it will receive the the new `User` object that was created as a result of the event, and can react to it accordingly.

2.2 AWS AppSync

AppSync is an AWS service designed to allow developers to build collaborative, data driven mobile and web applications. AppSync is essentially a Backend as a Service (BaaS) for GraphQL, packaged with several other useful features for development. AppSync manages everything needed to store, process, and retrieve the data for your application. Some useful features include:

- **Rapid GraphQL development**. AppSync provides a built-in GraphQL API builder as well as a query editor for writing, validating, and testing GraphQL operations, including queries, mutations and subscriptions [2]. This is a very useful tool for prototyping and testing APIs as you build them.
- **Autogenerated resolvers**. As mentioned above, a nice feature of GraphQL is the ability to interface with multiple different data sources from a single endpoint. AppSync has wizards for creating DynamoDB, Elasticsearch, Lambda and HTTP endpoint data sources, and *automatically generates the resolvers for them*, which significantly improves developer productivity.
- **Real time and offline data management**. The use of GraphQL subscriptions means that data can be shared and updated in real time across the application. Even when a user's device is offline, the data is cached locally and then synchronised later to AppSync when the device reconnects. This makes it perfect for collaborative applications such as chat or messaging apps, and the app that will be built in this demo.
- **Identity and Access Management**. AppSync is able to integrate with AWS Identity and Access Management (IAM) and Amazon Cognito for enabling application security and authentication.

2.3 AWS Amplify

Amplify is another service designed by AWS for developing cloud based mobile and web applications. Amplify provides toolchains, libraries and UI components as well as a serverless Lambda backend for application development. Some of the modules in the Amplify library include:

- **Authentication and Authorisation with Cognito**. Amplify uses Cognito User Pools to store user and information and to authenticate users to the application. It also uses Cognito Federated Identities to manage user acces to AWS resources.
- **Messaging queue integration**. Amplify's PubSub module allows you to integrate messaging between your application and its backend, using the MQTT messaging protocol.
- **Push Notifications**. The Push Notifications module allows you to integrate push notifications into your app using Amazon Pinpoint. Pinpoint allows you to send notifications, emails or messages to selected users of your application, as well as collect metrics about how users interact with the application.

While AppSync and Amplify can be used separately, used together they can provide a powerful ecosystem for increasing application development productivity. In particular, the Amplify CLI can be used to generate an AppSync GraphQL API. The Amplify GraphQL also provides an additional layer of simplicity compared to the AppSync client, and will be used in the practical section below.

2.4 Vue.js

Vue.js is a progressive Javascript frontend framework, which has increasingly gained popularity in the last two years, becoming one of the fasted growing tags on Stack Overflow [1]. Vue is a lightweight, modular framework which makes it incrementally adoptable, and only needs one Javascript file to be included for getting started.

The templating syntax for Vue is straightforward. A `.vue` component contains three main sections:

- **Template**. The template is essentially the view for the component and is delimited by the `<template>..</template>` tags. Variables from the script section can be interpolated in the view using the double brace syntax.
- **Script**. The script section contains the controller logic, and is delimited by the `<script>..</script>` tags.
- **Style**. Finally, the styles section contains any CSS styling for the component. It uses the `<style>..</style>` tags.

Below is an example of a Vue component for fetching and displaying a list of users (using the amplify GraphQL client):

```
<template>
  <div class="users">
    <div v-for="user in users" :key="user.id">
      {{ user.name }}
    </div>
  </div>
</template>

<script>
import { API, graphqlOperation } from "aws-amplify";

const ListUsers = `
  query {
    listUsers {
      items {
        id name email
      }
    }
  }
`;

export default {
  name: "ListUsers",
  data() {
    return {
```

```
      users: []
    };
  },
  async beforeCreate() {
    const data = await API.graphql(graphqlOperation(ListUsers));
    this.users = data.data.listUsers.items;
  }
};
</script>

<style>
.users {
  text-align: center;
  color: #eee;
}
</style>
```

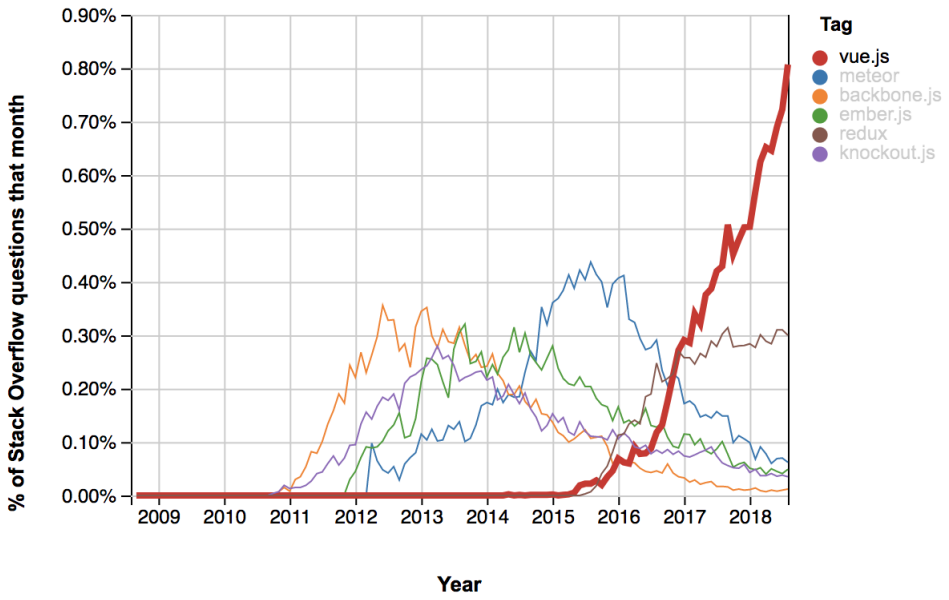


Figure 3: Vue.js adoption trend (Stack Overflow, 2018)

3 Practical Part 1: Application Skeleton

The first part of the practical will illustrate the process that was followed for setting up a Vue.js app with Amplify and AppSync integration. The steps for this process are:

1. Create the basic skeleton of the frontend web application using the Vue.js CLI.
2. Initialise Amplify from within the new project
3. Create a new AppSync GraphQL API using the Amplify CLI.

3.1 Scaffolding the Vue app

Vue provides an official CLI for quickly scaffolding Vue apps, similar to `create-react-app`. This can be installed using the `npm` package manager:

```
# Install the cli
npm install -g @vue/cli-service-global

# Create a new project
vue create jukebox
```

The `vue create` command opens up a prompt in the terminal to either use the default project settings or manually select features for the new project skeleton. Choosing manual selection allows finer grain control over the project template, and allows for additional features to be added (such as unit testing setup) that wouldn't otherwise be available.

```
Vue CLI v3.0.4
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, Router, Linter, Unit
? Use history mode for router? (Requires proper server setup for index fallback in production) Yes
[? Pick a linter / formatter config: Airbnb
? Pick additional lint features: Lint on save
? Pick a unit testing solution: Jest
? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? In package.json
? Save this as a preset for future projects? (y/N)
```

Figure 4: Vue project manual configuration

This provides:

- Babel configuration
- Vue Router for application routing
- Linter based on the Airbnb pattern (this is purely based on personal preference)
- Unit testing support with Jest (again, personal preference)

3.2 Initialising Amplify

Amplify needs to be configured and set up in order to start using the services (such as the GraphQL client) that it provides.

The `amplify configure` command is used to interactively configure Amplify from within the app:

```
[➜ jukebox git:(master) amplify configure                                    ]
Follow these steps to set up access to your AWS account:

Sign in to your AWS administrator account:
https://console.aws.amazon.com/
Press Enter to continue

Specify the AWS Region
? region:  eu-west-1
Specify the username of the new IAM user:
[? user name:  jukebox-amplify-user                                          ]
Complete the user creation using the AWS console
https://console.aws.amazon.com/iam/home?region=undefined#/users$new?step=final&accessK
ey&userNames=jukebox-amplify-user&permissionType=policies&policies=arn:aws:iam::aws:po
licy%2FAdministratorAccess
Press Enter to continue
```

The dialog asks for the AWS region, which should be set to `eu-west-1` for Ireland. It also allows you to create a new IAM user with the permissions needed by the application to interact with AWS. IAM is a service by AWS which defines access control lists for users and groups in your AWS account. In the AWS console, the `AdministratorAccess` policy needs to be attached to the new IAM role. This allows the IAM user full access to AWS services, which will be needed in the application as multiple services are used throughout.

		Policy name	Type	Used as	Description
☑	▸	AdministratorAccess	Job function	*None*	Provides full access to AWS services and re…

The new IAM role can now be used to create a new named profile. Profiles are a helpful tool which allow you to switch between different regions, credentials and IAM users. The profile name for this application will be `jukebox-profile`.

```
Enter the access key of the newly created user:
[? accessKeyId:  AKIAIYHEEI**********
[? secretAccessKey:  IvaA3Q+lXLEUQJe/ffQ9**********************
This would update/create the AWS Profile in your local machine
? Profile Name:  jukebox-profile
```

Once the profile has been created, the `amplify init` command can be executed to initialise Amplify in the new project:

```
[➜ jukebox git:(initial) amplify init
Note: It is recommended to run this command from the root of your app directory
? Choose your default editor: Visual Studio Code
? Choose the type of app that you're building javascript
Please tell us about your project
? What javascript framework are you using vue
[? Source Directory Path:  src
[? Distribution Directory Path: dist
[? Build Command:  npm run-script build
[? Start Command: npm run-script serve
Using default provider awscloudformation

For more information on AWS Profiles, see:
https://docs.aws.amazon.com/cli/latest/userguide/cli-multiple-profiles.html

[? Do you want to use an AWS profile? Yes
? Please choose the profile you want to use
  default
  boto3
❯ jukebox-profile
```

```
⠏ Initializing project in the cloud...

CREATE_IN_PROGRESS jukebox-20181007171214 AWS::CloudFormation::Stack Sun Oct 07 2018 17:12:15 GMT+0100 (IST) User Initiated
CREATE_IN_PROGRESS AuthRole               AWS::IAM::Role            Sun Oct 07 2018 17:12:17 GMT+0100 (IST)
CREATE_IN_PROGRESS UnauthRole             AWS::IAM::Role            Sun Oct 07 2018 17:12:18 GMT+0100 (IST)
CREATE_IN_PROGRESS DeploymentBucket       AWS::S3::Bucket           Sun Oct 07 2018 17:12:18 GMT+0100 (IST)
CREATE_IN_PROGRESS AuthRole               AWS::IAM::Role            Sun Oct 07 2018 17:12:18 GMT+0100 (IST) Resource creation Initiated
CREATE_IN_PROGRESS UnauthRole             AWS::IAM::Role            Sun Oct 07 2018 17:12:18 GMT+0100 (IST) Resource creation Initiated
CREATE_IN_PROGRESS DeploymentBucket       AWS::S3::Bucket           Sun Oct 07 2018 17:12:19 GMT+0100 (IST) Resource creation Initiated
✔ Successfully created initial AWS cloud resources for deployments.

Your project has been successfully initialized and connected to the cloud!

Some next steps:
"amplify status" will show you what you've added already and if it's locally configured or deployed
"amplify <category> add" will allow you to add features like user login or a backend API
"amplify push" will build all your local backend resources and provision it in the cloud
"amplify publish" will build all your local backend and frontend resources (if you have hosting category added) and provision it in the cloud

Pro tip:
Try "amplify add api" to create a backend API and then "amplify publish" to deploy everything


CREATE_COMPLETE AuthRole               AWS::IAM::Role            Sun Oct 07 2018 17:12:35 GMT+0100 (IST)
CREATE_COMPLETE UnauthRole             AWS::IAM::Role            Sun Oct 07 2018 17:12:35 GMT+0100 (IST)
CREATE_COMPLETE DeploymentBucket       AWS::S3::Bucket           Sun Oct 07 2018 17:12:39 GMT+0100 (IST)
CREATE_COMPLETE jukebox-20181007171214 AWS::CloudFormation::Stack Sun Oct 07 2018 17:12:42 GMT+0100 (IST)
```

This does the following:

- Creates auth and unauth roles in IAM
- Creates an S3 bucket for deploying the applications
- Creates a stack in cloud formation

3.3 Creating the API

Now that Amplify is properly set up and configured, new Amplify services can be added to the application. To add services to the application from the Amplify CLI, the `add` keyword is used. Adding the `API` service takes you through a guided process to create a hosted GraphQL server and API for your application through AWS AppSync:

```
➜  jukebox git:(master) ✗ amplify add api
[? Please select from one of the below mentioned services GraphQL
? Provide API name: jukebox
[? Choose an authorization type for the API Amazon Cognito User Pool
Using service: Cognito, provided by: awscloudformation
[ The current configured provider is Amazon Cognito.
 Do you want to use the default authentication and security configuration? Yes, use the default c
onfiguration.
Successfully added auth resource
? Do you have an annotated GraphQL schema? No
? Do you want a guided schema creation? true
? What best describes your project: (Use arrow keys)
❯ Single object with fields (e.g., "Todo" with ID, name, description)
  One-to-many relationship (e.g., "Blogs" with "Posts" and "Comments")
  Objects with fine-grained access control (e.g., a project management app with owner-based autho
rization)
```

For now, two simple Types will be added to the GraphQL schema: a `Playlist` and a `Song`:

```
type Playlist {
  id: ID!   // The exclamation mark means the field cannot be null
  name: String!
  description: String
  songs: [Song]
}

type Song {
```

```
  id: ID!
  name: String!
  artist: String
  album: String
  playlistId: ID
}
```

Once the Playlist type has been created and the API has been generated, the application is ready to be pushed. Performing an `amplify push` will build the API, generate all the neccessary resolvers (by default the schema and other data will be hosted in a DynamoDB table) and push it to AWS Appsync.

```
[➜  jukebox git:(amplify) ✗ amplify push
 | Category | Resource name | Operation | Provider plugin   |
 | -------- | ------------- | --------- | ----------------- |
 | Api      | jukebox       | Create    | awscloudformation |
 ? Are you sure you want to continue? (Y/n)
```

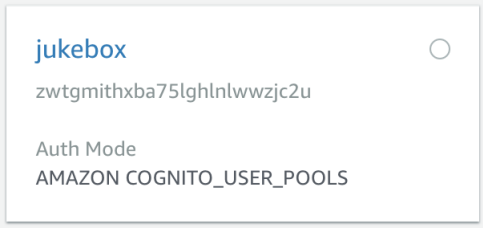


Figure 5: The API in the AppSync console

4 Practical Part 2: Queries, Mutations, Subscriptions

In the next part of the practical, a new Amplify GraphQL client will be created and used to fetch and modify data in the DynamoDB tables via the AppSync backend.

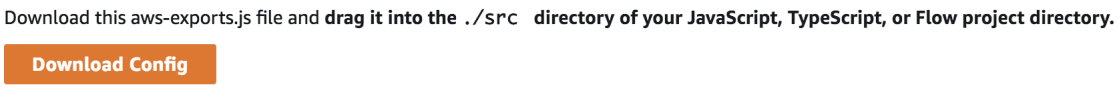
4.1 Amplify GraphQL Client

Firstly, the app needs to be configured to use AWS Amplify, so that the Amplify GraphQL client can be made available. To do this:

- The `aws-amplify` package must be installed:

```
npm install -save aws-amplify
```

- The `aws-exports` file for the project must be downloaded from the AppSync console:



- Finally, Amplify must be imported and configured with the previously downloaded `aws-exports` file:

```
// Client setup in main.js
import Amplify from "aws-amplify";
import config from "./aws-exports";
Amplify.configure(config);
```

Once Amplify is configured, the GraphQL client may be used anywhere by importing the `API` package from `aws-amplify`:

```
import { API, graphqlOperation } from "aws-amplify";

...
    const data = await API.graphql(graphqlOperation(ListPlaylists));
    this.playlists = data.data.listPlaylists.items;
...
```

The `aws-exports` file contains information about the GraphQL endpoint, the AWS region and auth for the application:

```
// WARNING: DO NOT EDIT. This file is automatically generated by AWS Amplify. It will be
    overwritten.

const awsmobile =  {
    "aws_appsync_graphqlEndpoint": "https://t2by4u6atva3hphxkbzzserv6m.appsync-api.eu-
        west-1.amazonaws.com/graphql",
    "aws_appsync_region": "eu-west-1",
    "aws_appsync_authenticationType": "API_KEY",
    "aws_appsync_apiKey": "******",
};

export default awsmobile;
```

4.2 Continuous Integration

At this point in the project, it would be nice to set up CI/CD to continuously build and deploy the project. For example, the suite of Jest unit tests could be run every time a new pull request is made against the `Jukebox` Github repository, to ensure that any new code that has been added has not broken the application. To do this, a free Continuous Integration platform called `CircleCI` will be used.

First CircleCI must be added to the list of Authorised OAuth Apps in Github. Then, from the CircleCI console, click "Set Up Project" for the desired repo:

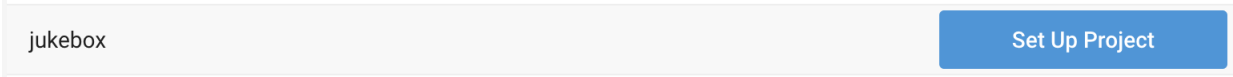


Figure 6: Set up new CircleCi project

Next, a folder called `.circleci` needs to be created in the root directory of the project and pushed to Github, with a `config.yml` file detailing the build instructions. The main step that needs to be added to the config file is the command for running the tests (the full config file can be found here, on Github):

```
# run tests
- run: npm run test:unit
```

CircleCI should now be properly configured and the build should be passing.

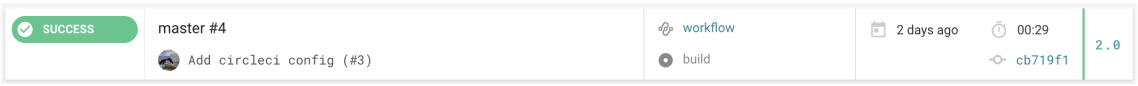


Figure 7: CircleCI successful build

This is confirmed by a green tick in the repository's commit history:

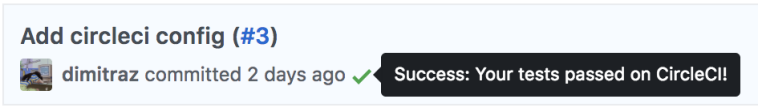


Figure 8: Commit history

4.3 Fetching data (Queries)

Now that the GraphQL client has been configured, we can start to fetch data from our DynamoDB tables through AppSync. The process for this is the following:

1. Define the GraphQL query for fetching songs from a given playlist using filters.
2. In Vue's component `created()` lifecycle event, use the query to make a call to the API and receive the data. Because this is an asynchronous call, the `async` keyword must be added to the `created()` function (alternatively, promises could be used to make these calls):

```
const data = await API.graphql(
  graphqlOperation(ListSongs(this.$route.params.id))
);
```

The final code for the `Playlist.vue` component looks like this:

```
import { API, graphqlOperation } from "aws-amplify";

const ListSongs = id => {
  return `query listSongs {
    listSongs(filter: {playlistId: {eq: "${id}" }}) {
      items {
        id
        name
        artist
        album
      }
    }
    }`;
};

export default {
  name: "Playlist",
  data() {
    return {
      songs: []
    };
  },
  async created() {
    const data = await API.graphql(
      graphqlOperation(ListSongs(this.$route.params.id))
    );
    this.songs = data.data.listSongs.items;
  }
};
```

4.3.1 A second iteration

Although the current component looks fine and is working as expected, one of the advantages of AppSync is that it can generate queries, mutations, subscriptions and resolvers for you. To generate these, the `amplify push api` command can be used. This command rebuilds the API/schema, generates the code locally and pushes it up to AppSync.

The second iteration of the `Playlist.vue` component looks a lot cleaner:

```
import { API, graphqlOperation } from "aws-amplify";
import { ListSongs } from "../graphql/queries";

export default {
  name: "Playlist",
  data() {
    return {
      songs: []
    };
  },
  async created() {
    const data = await API.graphql(
      graphqlOperation(ListSongs, { filter: {playlistId: {eq: "${this.$route.params.id}"
          }}})
    );
    this.songs = data.data.listSongs.items;
  }
};
```

4.3.2 A third iteration

This is looking a lot cleaner, but it would be nice to display information about the playlist and the playlist's songs in the component view without having to make separate API calls. Because of the way the schema is modelled, currently this is not possible. It also makes adding, deleting and updating songs within the playlist an almost impossible task without more than one API call and very tedious modification of the GraphQL DynamoDB resolvers.

To modify the GraphQL schema, the `amplify > backend > api > schema.graphql` file must be modified in the project folder. This is the updated schema:

```
type Playlist @model {
  id: ID!
  name: String!
  songs: [Song] @connection(name: "PlaylistSongs")
  createdAt: String
  description: String
}

type Song @model {
  id: ID!
  name: String!
  playlist: Playlist @connection(name: "PlaylistSongs")
  artist: String
  album: String
  genre: String
}
```

The `@model` object annotations mean that these objects are top-level entities in the generated API and will be stored in separate DynamoDB tables. The `@connection` annotation defines a relationship between two objects, and is the key difference compared to the last schema. Because this annotation has been added, AppSync will generate the resolvers needed to maintain this relationship. Additionally, a `createdAt` field was added, which will be autofilled by DynamoDB whenever a new playlist is created. For AppSync to take the changes, a `amplify push api` is run, which also generates updated mutations, queries and subscriptions locally to use within the app.

Now that the new schema has been generated, it makes fetching data about a playlist a lot more straightforward. Instead of using the `ListSongs` and `GetPlaylist` queries, a single `GetPlaylist` query can be used. The updated component looks like this:

```
import { Auth, API, graphqlOperation } from "aws-amplify";
import { GetPlaylist } from "../graphql/queries";

export default {
  name: "Playlist",
  data() {
    return {
      songs: [],
      playlist: {}
    };
  },
  methods: {
    async created() {
        const playlist = await API.graphql(
          graphqlOperation(GetPlaylist, { id: this.$route.params.id })
        );
        this.playlist = playlist.data.getPlaylist;
        this.songs = this.playlist.songs.items;
    }
  }
}
```

4.4 Modifying data (Mutations)

The code for executing mutations against the GraphQL API is very similar to the code for making queries. The following example shows how a song would be deleted from a playlist:

```
<template>
  <div class="container">
    <div v-for="song in songs" :key="song.id">
      Song name: {{ song.name }} <br>
      Song id: {{ song.id }} <br>
      Song artist: {{ song.artist }}<br>
      Song album: {{ song.album }}<br>

      <a @click="removeSong(song.id)">Remove</a>
    </div>
  </div>
</template>

<script>
import { Auth, API, graphqlOperation } from "aws-amplify";
import { GetPlaylist } from "../graphql/queries";
import { DeleteSong } from "../graphql/mutations";

export default {
  name: "Playlist",
  data() {
    return {
      songs: [],
      playlist: {}
    };
  },
  methods: {
    removeSong: async function(id) {
      const song = await API.graphql(
        graphqlOperation(DeleteSong, { input: { id: id } })
      );
      const songId = song.data.deleteSong.id;
      this.songs.splice(this.songs.findIndex(i => i.id === songId), 1);
    }
  },
  async created() {
    const playlist = await API.graphql(
      graphqlOperation(GetPlaylist, { id: this.$route.params.id })
    );
    this.playlist = playlist.data.getPlaylist;
    this.songs = this.playlist.songs.items;
  }
}
</script>
```

The process for removing a song is as follows:

1. First, import the `DeleteSong` mutation that was generated automatically by Amplify.
2. Next, add a new method called `removeSong(id)`. This is an asynchronous function that takes in the song ID and makes a call to the GraphQL API to delete the song with that ID. The `songs` array is also updated to remove the song for a "snappy ui" feel.
3. When iterating through the list of songs in the view template, add a link to remove the song which calls the new function using Vue's `@click` directive.

4.5 Realtime data (Subscriptions)

One of the main reasons for using GraphQL is the ease with which TCP Sockets can be used within your app using subscriptions. When implementing subscriptions, the app will get notified whenever an event

is triggered (for example, if a song is deleted), and can react to this immediately, in realtime. This is perfect for a realtime playlist scenario, where multiple users are adding songs to a playlist. They would expect to see new items appear without having to manually refresh the page.

The final playlist component with subscriptions looks like this:

```
import { Auth, API, graphqlOperation } from "aws-amplify";
import { GetPlaylist } from "../graphql/queries";
import { DeleteSong } from "../graphql/mutations";
import { OnCreateSong, OnDeleteSong } from "../graphql/subscriptions";

export default {
  name: "Playlist",
  data() {
    return {
      songs: [],
      playlist: {}
    };
  },
  methods: {
    removeSong: async function(id) {
      const song = await API.graphql(
        graphqlOperation(DeleteSong, { input: { id: id } })
      );
      const songId = song.data.deleteSong.id;
      this.songs.splice(this.songs.findIndex(i => i.id === songId), 1);
    }
  },
  async created() {
    const playlist = await API.graphql(
      graphqlOperation(GetPlaylist, { id: this.$route.params.id })
    );
    this.playlist = playlist.data.getPlaylist;
    this.songs = this.playlist.songs.items;

    // Subscribe to creation of a song
    const subscriptionCreate = API.graphql(
      graphqlOperation(OnCreateSong, { songPlaylistId: this.$route.params.id })
    ).subscribe({
      next: song => {
        this.songs.push(song.value.data.onCreateSong);
      }
    });

    // Subscribe to deletion of a song
    const subscriptionDelete = API.graphql(
      graphqlOperation(OnDeleteSong, { songPlaylistId: this.$route.params.id })
    ).subscribe({
      next: song => {
        const songId = song.value.data.onDeleteSong.id;
        this.songs.splice(this.songs.findIndex(i => i.id === songId), 1);
      }
    });
  }
};
```

The code includes subscriptions that will be triggered any time a song is added or deleted from the playlist. The process for including subscriptions in the component is as follows:

1. First, import the `OnCreateSong` and `OnDeleteSong` subscriptions that were generated by Amplify.
2. Next, in the `created()` lifecycle method, subscribe to the OnCreateSong and OnDeleteSong observables, filtering by songs that were added or created from the specific playlist. Once any new `Song` object comes in, add it or delete it from the `songs` array, again for a snappy ui feel.

5 Practical Part 3: Offline syncing

The third part of the practical will illustrate how to handle requests (such as queries and mutations) within the app when a user is offline. The structure for this part of the practical is broken down into the sequence:

1. Create a new AppSync GraphQL client based on Vue Apollo
2. Replace existing queries, mutations and subscriptions from the Amplify client with calls to the AppSync client
3. Execute queries, mutations and subscriptions while offline, and see them sync up when back online

5.1 AppSync Client with Vue Apollo

Offline mutations are a property of the GraphQL client being used. In this case, the Amplify client does not support offline mutations, while the AppSync client is built on top of Apollo, which has built-in support for offline synching. Unfortunately this means the current Amplify client and its API calls must be replaced with AppSync/Vue Apollo calls, which is a time-consuming task (and produces lengthier code), but will allow the advantage of offline support.

Firstly, the `aws-appsync` and `vue-apollo` packages need to be installed:

```
npm i -save aws-appsync vue-apollo
```

Next, the `main.js` and `App.vue` files need to be modified to configure the client:

```
// main.js
..
import AWSAppSyncClient from "aws-appsync";
import VueApollo from "vue-apollo";
Vue.use(VueApollo);
..

const client = new AWSAppSyncClient(
  {
    url: config.aws_appsync_graphqlEndpoint,
    region: config.aws_appsync_region,
    auth: {
      // Amazon Cognito user pools using AWS Amplify
      type: config.aws_appsync_authenticationType,
      jwtToken: async () =>
        (await Amplify.Auth.currentSession()).getIdToken().getJwtToken()
    }
  },
  {
    defaultOptions: {
      watchQuery: {
        fetchPolicy: "cache-and-network"
      }
    }
  }
);

const appsyncProvider = new VueApollo({
  defaultClient: client
});

new Vue({
  router,
  provide: appsyncProvider.provide(),
  render: h => h(App)
}).$mount("#app");
```

```
// App.vue
<script>
export default {
  name: "App",
  data: () => ({ hydrated: false }),
  async mounted() {
    await this.$apollo.provider.defaultClient.hydrated();
    this.hydrated = true;
  }
};
</script>
```

5.2 Queries

Now that the AppSync client is configured, the existing API calls can be switched out to make use of the new client:

```
import { Auth, API, graphqlOperation } from "aws-amplify";
import { GetPlaylist } from "../graphql/queries";

export default {
  name: "Playlist",
  data() {
    return {
      playlist: {},
      playlistSongs: []
    };
  },
  apollo: {
    playlist: {
      query: () => GetPlaylist,
      update: data => data.getPlaylist,
      variables() {
        return {
          id: this.$route.params.id
        }
      }
    }
  }
};
```

Each query declared in the `apollo` object in a component results in the creation of a smart query object. In this example, this means that Apollo will automatically execute the query labelled with `playlist` when the component is created, and it will additionally update the corresponding `playlist` field in the component with the new playlist data that was fetched. This is done in the `update` method. The `variables` method returns the parameters that should be used when executing the query. Queries will automatically be cached when offline and executed when the user comes back online, so no additional set up for offline synching is necessary.

5.3 Mutations

Executing mutations is slightly more convoluted. New mutations need to be declared in new methods in the component, and make a call to the `this.$apollo.mutate` function:

```
import { Auth, API, graphqlOperation } from "aws-amplify";
import { GetPlaylist } from "../graphql/queries";
import { CreateSong, DeleteSong } from "../graphql/mutations";
```

```
export default {
  name: "Playlist",
  data() {
    return {
      playlist: {},
      playlistSongs: []
    };
  },
  methods: {
    addSong(song) {
      this.$apollo
        .mutate({
          mutation: CreateSong,
          variables: {
            input: {
             ..this.song,
              songPlaylistId: this.$route.params.id
            }
          },
          update: (store, { data: { createSong } }) => {
            // Read the data from the cache for the playlist
            const data = store.readQuery({
              query: GetPlaylist,
              variables: { id: this.$route.params.id }
            });

            // Add the new song to the playlist's song array
            data.getPlaylist.songs.items.push(createSong);

            // Write the playlist back to the cache
            store.writeQuery({
              query: GetPlaylist,
              variables: { id: this.$route.params.id }
            });
          },
          optimisticResponse: {
            __typename: "Mutation",
            createSong: {
              __typename: "Song",
              id: uuid4(),
             ..song
            }
          }
        })
        .then(data => console.log(data))
        .catch(error => console.error(error));
    }
  },
  apollo: {
    playlist: {
      query: () => GetPlaylist,
      update: data => data.getPlaylist,
      variables() {
        return {
          id: this.$route.params.id
        };
      }
    }
  }
};
```

The `$apollo.mutate` function takes in various parameters:

- `mutation`: The mutation query
- `variables`: The parameters to pass to the mutation when it is being executed

- `update`: The steps needed to update the cache when offline. The definition from the Apollo docs:

```
"This function will be called twice over the lifecycle of a mutation. Once at the
    very beginning if an optimisticResponse was provided. The writes created from
    the optimistic data will be rolled back before the second time this function is
    called which is when the mutation has succesfully resolved. At that point update
     will be called with the actual mutation result and those writes will not be
    rolled back."
```

- `optimisticResponse`: An optimistic response, used to simulate a zero latency server response. Because the `update` function makes two roundtrips, this is important to give the user immediate feedback. The optimistic response will contain the expected response from the server, and a random dummy ID for the object to avoid collisions. This ID will be replaced with an ID generated by DynamoDB once the mutation is actually executed on the server.

This is all the code that is needed to get offline mutations working - executing queries and mutations with the new client while offline shows an optimistic UI response at first, but will be replaced with actual objects once the user is back online and the queries are executed against the AppSync backend.

5.4 Subscriptions

Finally, subscriptions can be replaced by using the `subscribeToMore` keyword within the query definition. This is much cleaner and simpler than the mutation code:

```
apollo: {
    playlist: {
      query: () => GetPlaylist,
      update(data) {
        this.playlistSongs = _.uniqBy(data.getPlaylist.songs.items, "id");
        return data.getPlaylist;
      },
      variables() {
        return {
          id: this.$route.params.id
        };
      },
      subscribeToMore: {
        document: OnCreateSong,
        updateQuery(previousResult, { subscriptionData }) {
          const newSong = subscriptionData.data.onCreateSong;

          // If we added the song already don't do anything
          // This can be caused by the `updateQuery` of the addSong mutation
          if (
            previousResult.getPlaylist.songs.items.find(
              song => song.id === newSong.id
            )
          ) {
            return previousResult;
          }
          // Return the subscription data
          previousResult.getPlaylist.songs.items.push(newSong);
          this.playlistSongs = previousResult.getPlaylist.songs.items;
          return {
            playlist: previousResult.getPlaylist
          };
        }
      }
    }
}
```

The code for adding subscriptions is quite straightforward:

1. In the `document` field define the subscription query that should be used
2. In the `updateQuery` method define what should happen when an object comes in from a subscription. In this case the song is added to the playlist's list of songs if it isn't there already.

6 Practical Part 4: User Authentication with Cognito Pools

When creating the AppSync API, we specified that authentication to the API should occur using Cognito User Pools, and a new Pool was created for the app. To connect the application to the Cognito Pool, there are 2 steps that need to be performed:

1. Add `user` support to the API. This means that users can create and keep track of their own jukeboxes, and songs added to any jukebox will identify the user that added them.
2. Create a login/signup form that will keep track of the current user, making sure that users who are not authenticated don't have access to the app.

6.1 Updating the schema

For the first step, the schema needs to be edited again to accept a `username` field:

```
type Playlist @model {
  id: ID!
  name: String!
  songs: [Song] @connection(name: "PlaylistSongs")
  createdAt: String
  description: String
  username: String
}

type Song @model {
  id: ID!
  name: String!
  playlist: Playlist @connection(name: "PlaylistSongs")
  artist: String
  album: String
  genre: String
  username: String
}
```

Once this has been pushed, the DynamoDB mutation resolvers[1] must be edited to point to the currently logged in Cognito user. In particular, the `createPlaylist()` and `createSong()` resolvers need to be modified. AppSync makes an object called `identity` available within the `$context` object, which can be used to identify the current user:

```
$util.qr($context.args.input.put("userId", $context.identity.username))
```

The final resolver for `createPlaylist()` looks like this:

[1] Resolvers for DynamoDB are written in a templating language called Apache Velocity

```
## START: Prepare DynamoDB PutItem Request. **
$util.qr($context.args.input.put("createdAt", $util.time.nowISO8601()))
$util.qr($context.args.input.put("updatedAt", $util.time.nowISO8601()))
$util.qr($context.args.input.put("userId", $context.identity.username))
$util.qr($context.args.input.put("__typename", "Playlist"))
{
  "version": "2017-02-28",
  "operation": "PutItem",
  "key": {
      "id": {
          "S": "$util.autoId()"
    }
  },
  "attributeValues": $util.dynamodb.toMapValuesJson($context.args.input),
  "condition": {
      "expression": "attribute_not_exists(#id)",
      "expressionNames": {
          "#id": "id"
    }
  }
}
## END: Prepare DynamoDB PutItem Request. **
```

To test this change, first a new playlist will be created in the AppSync console. Note that the query returns the username field:

```
mutation CreatePlaylist {
  createPlaylist(input: {
    name: "Testing Username"
  }) {
    id
    name
    username
  }
}
```

This returns the following:

```
{
  "data": {
    "createPlaylist": {
      "id": "67fcf23c-de83-4f90-a7c4-11ccf0ccae6e",
      "name": "Testing Username",
      "username": "dimitra"
    }
  }
}
```

Next, a named query called `GetPlaylistsByUsername` is created to list playlists filtered by username:

```
query GetPlaylistsByUsername {
  listPlaylists(filter: {
    username: {
      eq: "dimitra"
    }
  }) {
    items {
      name
      description
      username
    }
  }
}
```

This is the result:


```
{
  "data": {
    "listPlaylists": {
      "items": [
        {
          "name": "hello",
          "description": null,
          "username": "dimitra"
        },
        {
          "name": "Testing Username",
          "description": null,
          "username": "dimitra"
        }
      ]
    }
  }
}
```

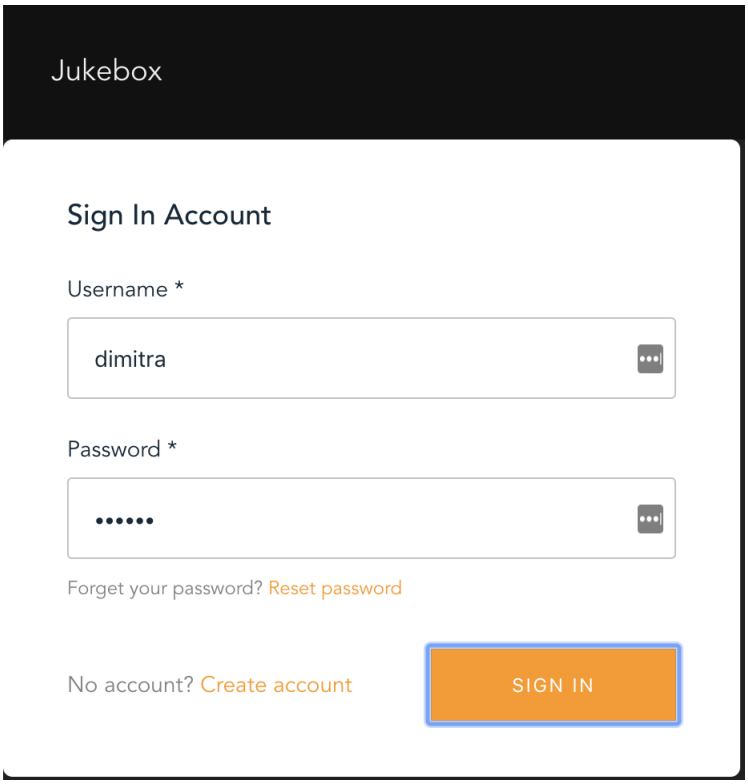

6.2 User authentication

Amplify offers several UI components that can be dropped into your application. To use these plugins, the Amplify import in `main.js` needs to be updated to look like so:

```
import Amplify, * as AmplifyModules from 'aws-amplify'
import { AmplifyPlugin } from 'aws-amplify-vue'
Amplify.configure(aws_exports)

Vue.use(AmplifyPlugin, AmplifyModules)
```

Then, the component can be used in the view template:



These components offer a nice option for quick prototyping, but are not very easily customised. For this reason they will not be used in the Jukebox application, and simple custom authentication components will be used instead, to match the look and feel of the app.

To create a custom login form, the `Auth` module from Amplify can be used to log in, sign up or get information about the currently signed in user. The `login()` function for `Login.vue` looks like this:

```
login() {
  Auth.signIn(this.username, this.password)
    .then(user => {
      localStorage.authenticated = true;
      this.$router.push({name: 'home'})
    })
    .catch(err => this.error = "Unable to log in");
}
```

This sets the variable `authenticated` in local storage to true, and redirects the user to the home page.

For guarding routes against unauthorised users, `router.js` needs to be updated to include middleware that checks if the user is authenticated or not. If `authenticated` is set to false or undefined in local storage, the user is redirected to the login page.

```
router.beforeEach((to, from, next) => {
  if (to.path !== "/login") {
    if (!localStorage.authenticated) {
      next({ name: "login" });
    } else {
      next();
    }
  } else {
    next();
  }
});
```

7 Practical Part 5: Sentiment Analysis with AWS Comprehend

The final part of the practical will include using the AWS Comprehend service to perform sentiment anaylsis on the lyrics of the Jukebox, in order to output the Jukebox's overall mood. To do this:

1. A `lyrics` field must be added to the Song schema, which is what will be used to perform the sentiment analysis on.
2. A `mood` field must be added to the Playlist schema, to keep track of the mood of the Jukebox.
3. A Lambda function will be created to take in the lyrics of all of the songs in the jukebox and send them to AWS Comprehend. To expose the Lambda function, an API Gateway endpoint will be set up that the Vue app will make a call to.

The final GraphQL schema looks like this:

```
type Playlist @model {
  id: ID!
  name: String!
  songs: [Song] @connection(name: "PlaylistSongs")
  createdAt: String
  description: String
  username: ID
  mood: String
}

type Song @model {
  id: ID!
  name: String!
  playlist: Playlist @connection(name: "PlaylistSongs")
  artist: String
  album: String
  genre: String
  username: ID
  lyrics: String
}
```

The lambda function is relatviely simple and looks like this:

```
import boto3, json

client = boto3.client('comprehend')

def respond(err, res=None):
    return {
        'statusCode': '400' if err else '200',
        'body': err.message if err else json.dumps(res),
        'headers': {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*'
        },
    }

def lambda_handler(event, context):
    """
    A function for detecting the overall sentiment of
    a Jukebox using Comprehend sentiment analysis
    """
    sentiment = client.detect_sentiment(Text=event['lyrics'], LanguageCode='en')['
        Sentiment']

    return sentiment
```

To be able to execute this function, we need to assign the Lambda the correct permissions (Full CloudWatch access for logs, full Comprehend access):

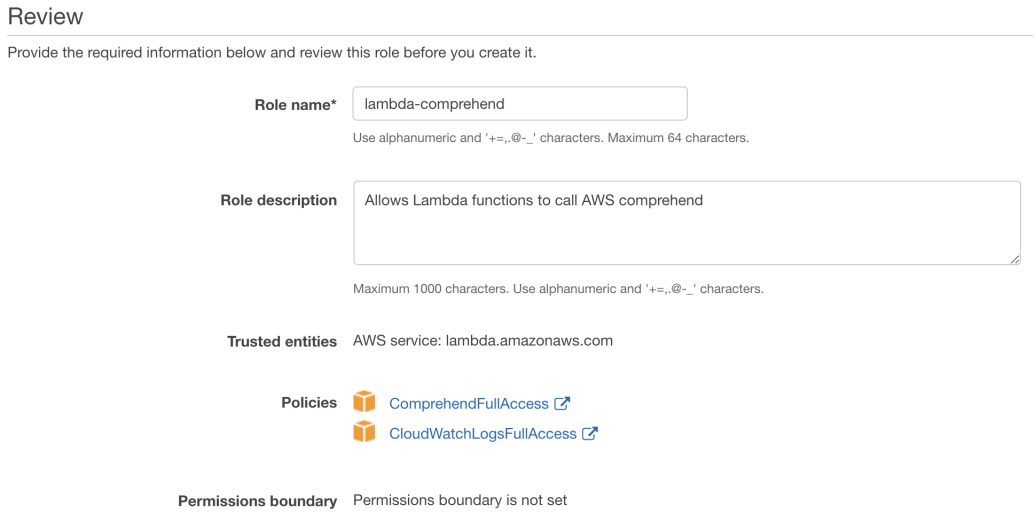


Figure 9: Lambda IAM roles

Next, to expose this Lambda function to our app, API Gateway will be used. In order to do this, a new API must be created. This API will be called `Jukebox`:

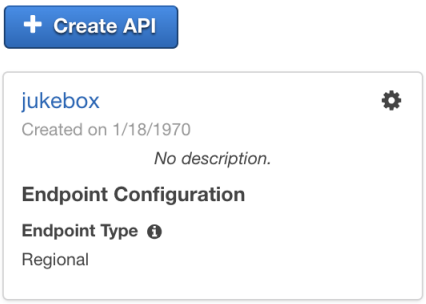


Figure 10: API Gateway resource

Next, a new resource called `songs` (this will be the actual endpoint path) with a new `POST` method must be created, making sure that CORS is enabled for local testing. The configuration for the `POST` method needs to point to the Lambda function, and looks like this:

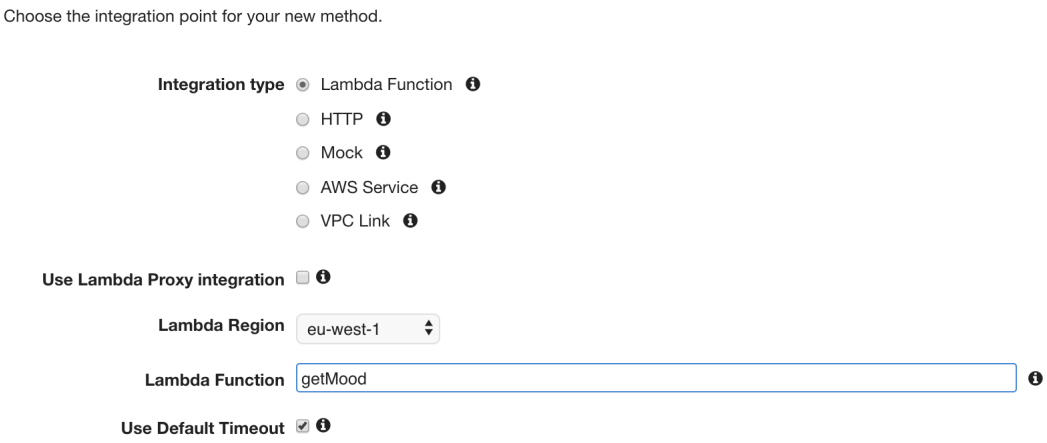


Figure 11: API Gateway resource POST endpoint

Finally, this endpoint can be called in the `Playlist.vue` component any time an update is made to the list of songs. The method for getting the mood looks like this:

```
getMood() {
  let lyrics = "";
  _.uniqBy(this.playlistSongs, "name").forEach(song => {
    if (song.lyrics) {
      lyrics += song.lyrics;
    }
  });

  if (lyrics) {
    axios
      .post(process.env.VUE_APP_ENDPOINT, {
        lyrics: lyrics
      })
      .then(res => {
        this.mood = res.data;
      })
      .catch(error => {
        console.log(error);
      });
  } else {
    this.mood = 'NEUTRAL';
  }
}
```

This code:

1. Gets the lyrics from each song in the playlist and appends them to a string called `lyrics`.
2. If the `lyrics` variable is not empty, a call will be made to the API Gateway endpoint (`POST https://skkhhvm8a3.execute-api.eu-west-1.amazonaws.com/prod/songs`), which will return a sentiment from the lyrics: NEUTRAL, POSITIVE, or NEGATIVE.
3. Otherwise, if the `lyrics` variable is empty, return a NEUTRAL mood.

This mood can now be used in the view template to display a nice smiley face icon corresponding to each mood:

```
<div>
Mood:
  <i v-if="mood === 'NEUTRAL'" class="far fa-meh"></i>
  <i v-if="mood === 'POSITIVE'" class="far fa-smile"></i>
  <i v-if="mood === 'NEGATIVE'" class="far fa-frown"></i>
</div>
```

Figure 12: The Jukebox's mood

8 The Final Jukebox Application

Everything is finished, and the website can now be pushed to the cloud. Luckily, Amplify makes it very simple to build and push the app to an S3 bucket, with the `hosting` feature:

```
➜  jukebox git:(master) ✗ amplify hosting add
? Select the environment setup: DEV (S3 only with HTTP)
? hosting bucket name jukebox-20181125171351--hostingbucket
? index doc for the website index.html
? error doc for the website index.html

You can now publish your app using the following command:
Command: amplify publish
```

Figure 13: Amplify add hosting

Once this is published, clicking on the S3 link will redirect you to the login page. A user can authenticate against their Cognito pool credentials. This functionality uses the `Auth` module from Amplify:

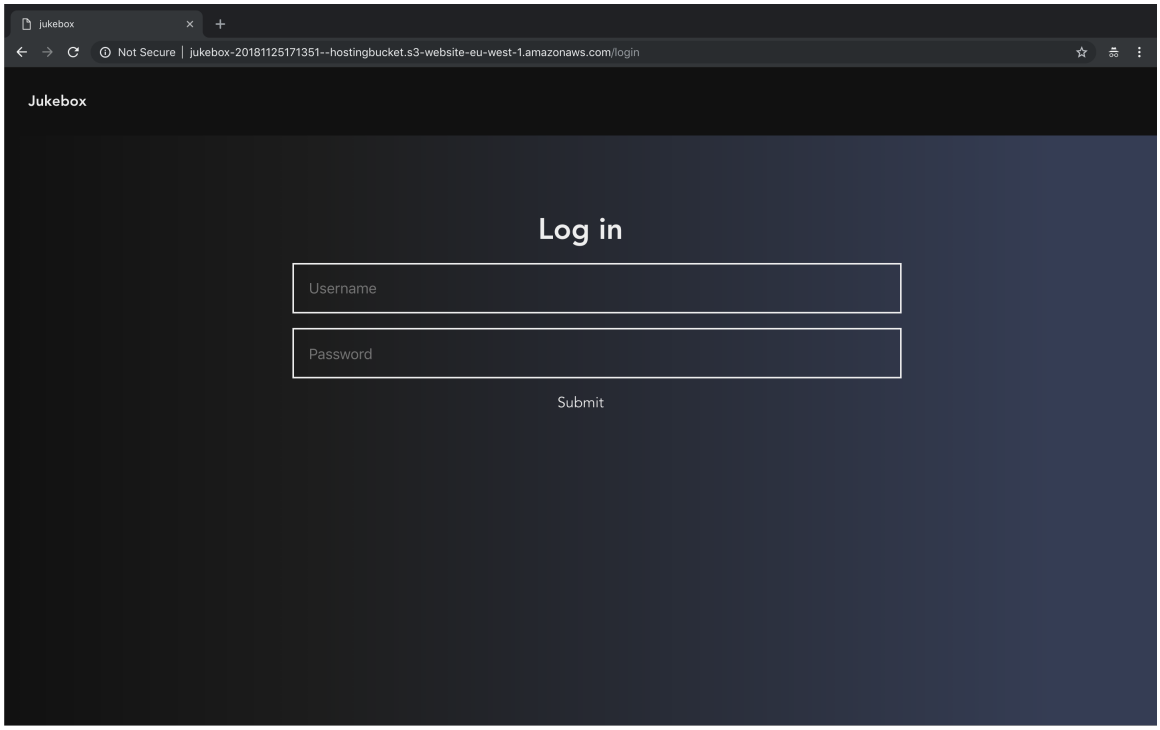


Figure 14: Logging in with Cognito

Next, a user will be displayed the list of jukeboxes that they've created or contributed to. This uses the `listPlaylists` GraphQL query, filtering results against the user's username:

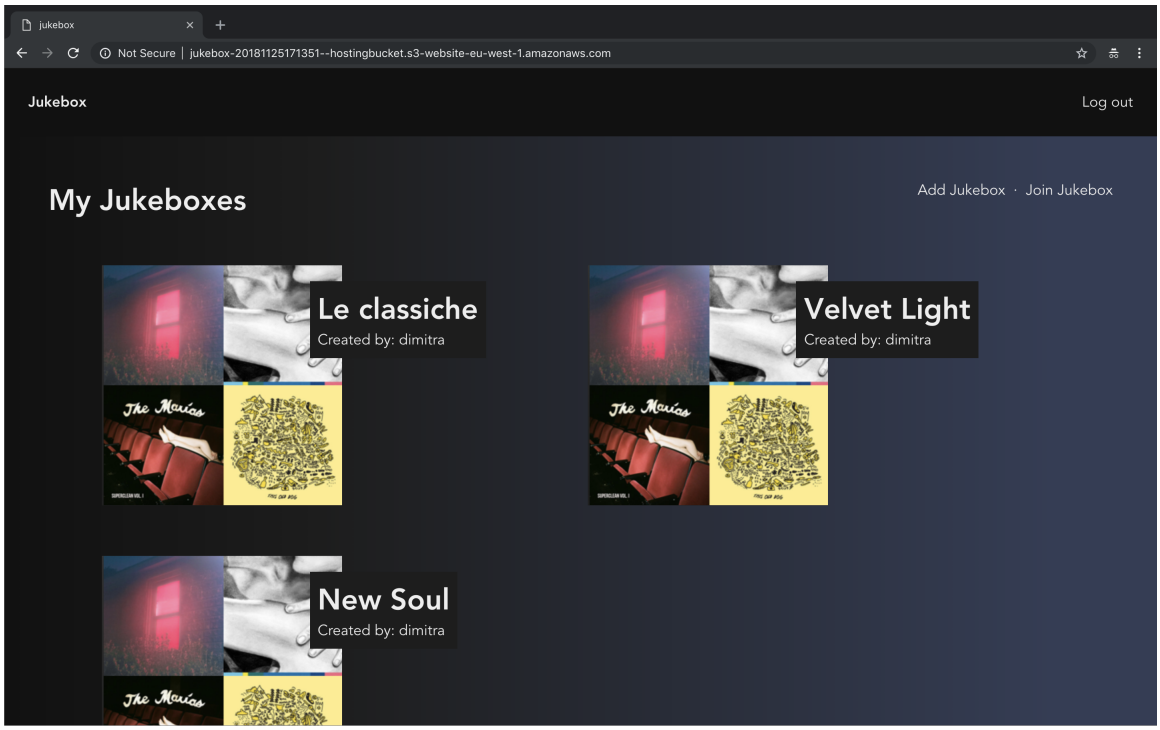


Figure 15: List of Jukeboxes

A user can create a new jukebox item. This uses the `createPlaylist` GraphQL mutation:

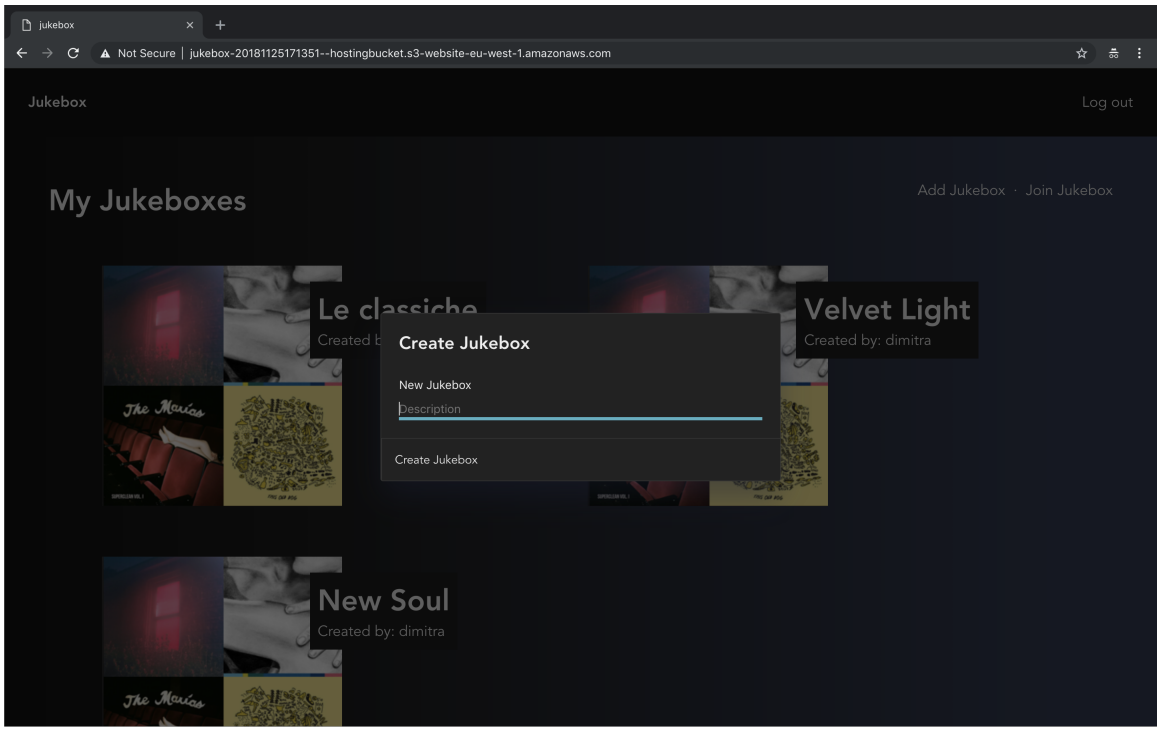


Figure 16: Create a new Jukebox modal

A user can join another user's jukebox and begin adding songs to it. This does not need any special query or mutation.

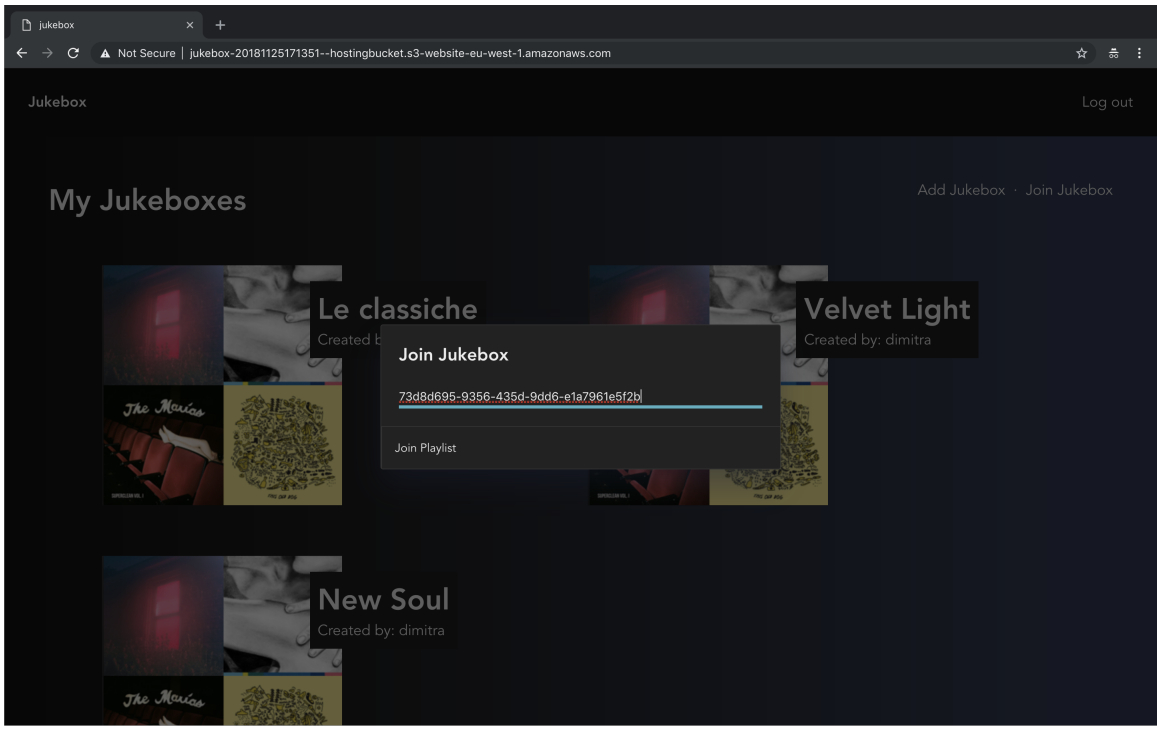


Figure 17: Join another user's Jukebox modal

A user can add a song to a playlist. This uses the `createSong` GraphQL mutation:

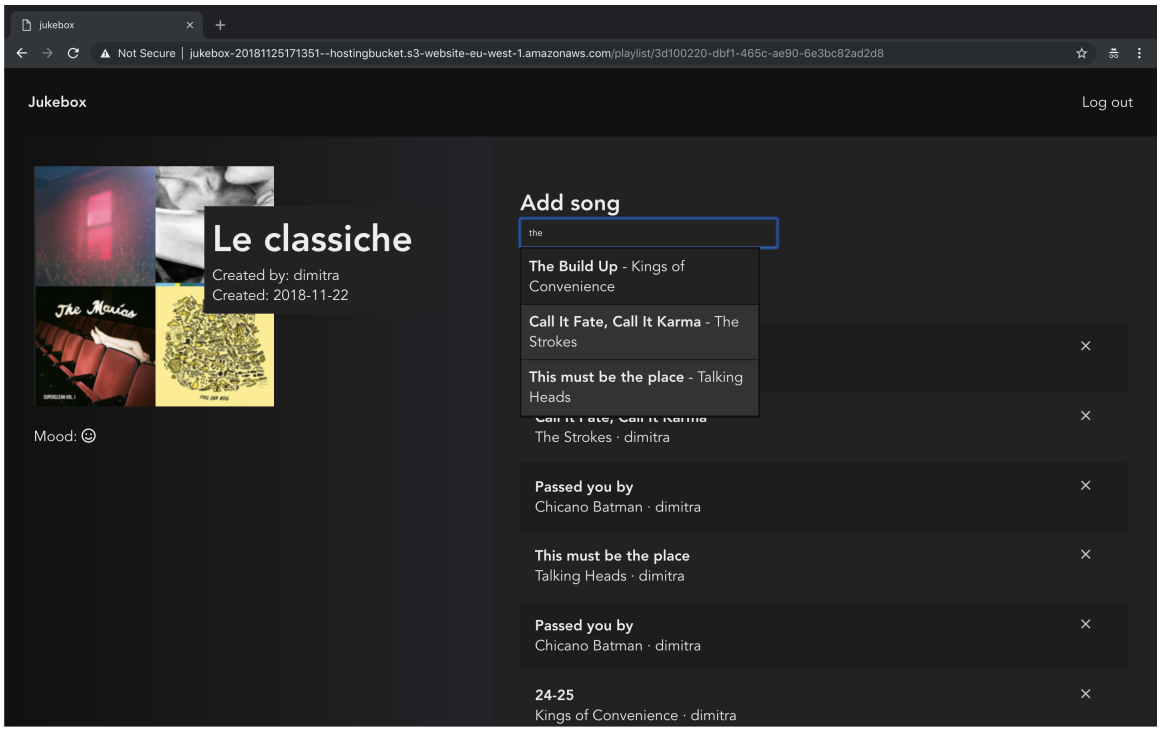


Figure 18: Add a song to a Jukebox

A user can receive realtime updates from other users who are adding songs to the same jukebox. This uses the `onCreateSong` GraphQL subscription:

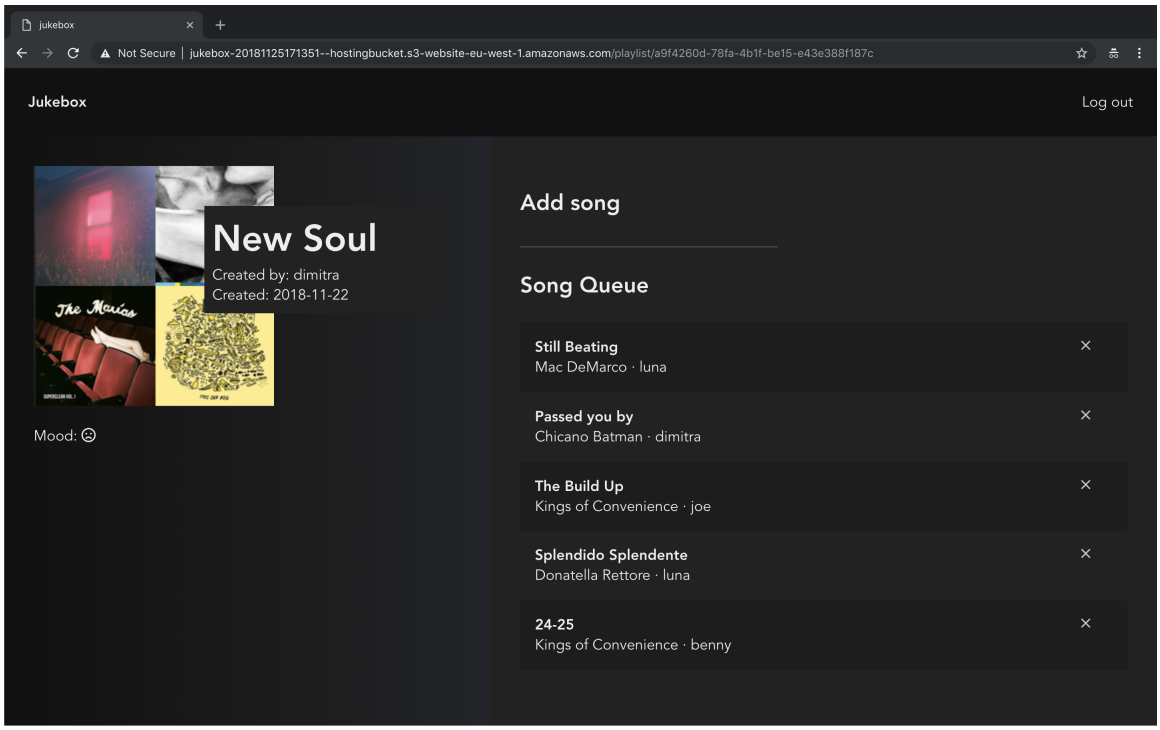


Figure 19: Multiple user collaboration

Note that users can add the same songs to the playlist. This is because each song created will have a unique ID, and this is how AppSync handles conflict resolution.

8.1 Improvements

The application developed in this report was a simple proof of concept to get practical, hands on experience with the services being explored. This means there is still a lot of room for improvement:

- It would be nice to include a modal in the Playlist component that allows a user to edit the playlist (name, description, album art). This would only need a simple `updatePlaylist(...)` mutation, with a function for hosting the uploaded image online, possibly to an S3 bucket in AWS. This didn't seem like an essential feature to include in the application prototype.
- For a more user friendly experience, a user should be able to copy a playlist's ID to the clipboard to share with friends, instead of having to do this manually.
- The `songs.json` should be hosted somewhere online and fetched on demand. For a nicer architecture, this could also be entirely replaced with another DynamoDB table, and each `Song` object could just contain a reference to the actual song in the database. Alternatively, this could also be replaced with a query to an actual Song API/Database.

8.2 Additional Resources

- Github repository: https://github.com/dimitraz/jukebox/
- Youtube walkthrough: https://www.youtube.com/watch?v=dFgxnveeeKE
- Realtime demonstration gif: https://raw.githubusercontent.com/dimitraz/jukebox/master/images/realtime.gif
- Adding/deleting songs demonstration gif: https://raw.githubusercontent.com/dimitraz/jukebox/master/images/add_delete.gif

9 Conclusions

While I enjoyed learning about and getting to work with these new technologies, I ran into many difficulties that I think should be noted:

- Documentation could be greatly improved. Because AppSync and Amplify are newer services, I found the documentation hard to use and follow, and lacking in several areas, which led me down rabbit holes that could have been avoided if properly documented. Even Github issues and Stackoverflow questions only seem to address problems for certain areas of the services, and are mainly based on integration with React. No information at all is available about usage of Vue Apollo with AppSync.

- AWS Amplify and AWS AppSync integration is not as seamless as I initially thought. The integration between them seems like it still has a way to go, and there are overlapping components, which causes confusion about when to use one over the other, when to use them together, etc. Again, documentation is severely lacking and a lot of the time I found myself aimlessly searching on Github for existing examples or direction.

- Because I chose a technology stack with technologies I had never used before, the learning curve was extremely steep. I struggled particularly to wrap my head around GraphQL schemas initially, and then DynamoDB resolvers, which were written in a templating language I couldn't understand. Again, accessible documentation for beginners was impossible to find, and the information I found was mostly pointed at users with a good knowledge in the area already.

A few takeaways I got from working on this term paper:

- I thoroughly enjoyed learning Vue.js and found the documentation to be simple and exhaustive. Vue.js seems easy to use and a much more concise alternative to other popular frameworks, such as React.

- I also enjoyed learning about GraphQL and drawing comparisons with Rest APIs. While I do see advantages in some areas, I noticed that the GraphQL queries could become very verbose (for example when writing Apollo mutations) and hard to keep track of, which is why I initially chose the Amplify client over the more popular Apollo client. I could see how these types of applications might become difficult to manage in enterprise contexts. However, I do think that an interesting case could be made for GraphQL vs Rest API.

References

[1] Ian Allen. The brutal lifecycle of javascript frameworks, Jan 2018.

[2] AWS Amplify. The foundation for your cloud-powered mobile web apps.

[3] How To GraphQL. Big picture (architecture).

[4] Sacha Greif. So what's this graphql thing i keep hearing about?, Apr 2017.

[5] Orinami Olatunji. Build a simple graphql api server with express and nodejs, May 2017.

Bibliography

- Creating AWS AppSync GraphQL APIs with AWS Amplify
  https://www.youtube.com/watch?v=p7mwQaGo6P0
- Vue.js Documentation
  https://vuejs.org/v2/guide/
- Vue.js CLI Documentation
  https://cli.vuejs.org/guide/
- AWS Amplify Documentation
  https://aws-amplify.github.io/
- How to Build Serverless Vue Applications with AWS Amplify
  https://hackernoon.com/how-to-build-serverless-vue-applications-with-aws-amplify-67d16c79e9d6
- Deploying Vue to Amazon S3 with CircleCI
  https://rossta.net/blog/deploying-vue-to-amazon-s3-with-circleci.html
- Introducing the AWS Amplify GraphQL Client
  https://hackernoon.com/introducing-the-aws-amplify-graphql-client-8a1a1e514fde
- Vue Apollo Documentation
  https://akryum.github.io/vue-apollo/guide/apollo/