Build a Full-Stack App - Ultimate Guide to AWS Amplify with React

freecodecamp.org/news/p/1c12dc9d-5979-425c-bb04-782e7650d5bb

AWS Amplify is a tool that has been developed by AWS to make App development easier. It includes loads of features which allow you to quickly and easily work with other AWS services. This means you can spend more time building the features that make your app unique.

This tutorial is split into 4 parts, creating our app with login, adding a database and working with the data, adding file storage and using those files, and then allowing users to upload their own files and data.

Creating Our App with Signup, Login and Logout

In this first section we'll be setting up a new React App with AWS Amplify to add Sign up, Login and Logout in the easiest way possible.

We need to start by creating a new react app using `create-react-app`. Open a terminal and run these commands. If you don't have create-react app installed then you can run `npm i -g create-react-app` first.

```
npx create-react-app amplify-react-app

cd amplify-react-app
```

With that set up we can now install Amplify and then configure it.

```
npm install -g @aws-amplify/cli

amplify configure
```

This will open up an AWS console tab in your browser. Make sure that you're logged into the correct account with a user that has admin permissions.

Go back to the terminal and follow the steps, adding a region and name for the user. This will then take you back the the browser where you can follow the steps to create the new user. Make sure to stay on the page where you see the key and secret!

Back in the terminal again you can follow the steps, copying the access key and secret into the terminal when asked. When you are asked if you want to add this to a profile say `Yes`. Create a profile that is something like `serverless-amplify`.

Now we can initialise the amplify setup by running `amplify init`. You can give the project a name and answer all the questions. Most of them should be correct already. This then takes a while to make the changes on your account.

Once done we need to add authentication to the app. We do this with `amplify add auth`. Select the method as `default` the sign in to `email` and then `no, I am done`. We can now deploy this by running `amplify push`. This takes a while but at the end, our `src/aws-exports.js` file has been created.

The React App

Now we can get onto creating the react app. Start by installing the Amplify npm packages we need.

```
npm install --save aws-amplify @aws-amplify/ui-react
```

Now we can start editing the code of our app. In our `src/App.js` file we can remove everything in the headers and replace it with this:

```
<header className="App-header">
   <AmplifySignOut />
   <h2>My App Content</h2>
</header>
```

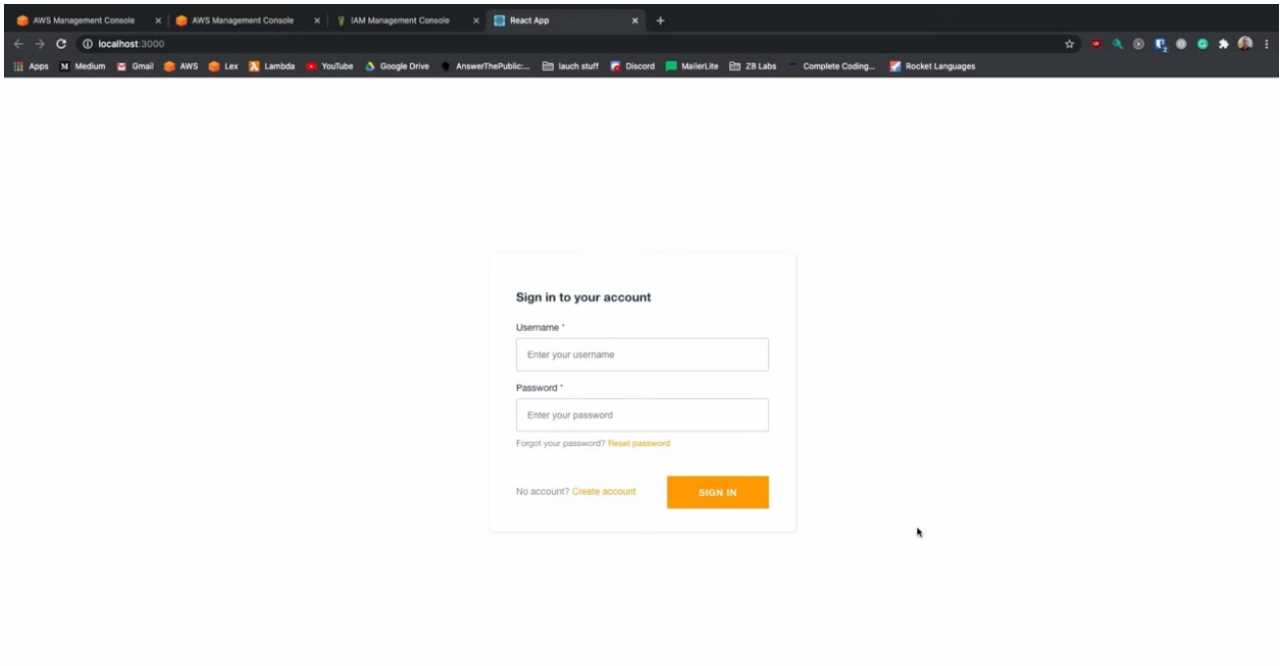
This is a very basic set up but you could put the main content of your site here and put the `AmplifySignOut` button where ever you want it to be.

We also need to add some extra imports to the top of the file:

```
import Amplify from 'aws-amplify';
import awsconfig from './aws-exports';
import { AmplifySignOut, withAuthenticator } from '@aws-amplify/ui-react';

Amplify.configure(awsconfig);
```

Now the last thing that we need to do is to change the way that we export the app. change the last line to be `export default withAuthenticator(App);` to add Amplify to this app.

Now when we run `npm start` we should get a login screen. We've not created this so it has come from Amplify it's self.



If we try and log in then it will fail as we need to sign up first. We can click `create account` and then enter our email and a password to sign up. Once we've confirmed our email by submitting the code we were sent we get onto the home page of our app. If we log out we can now log back in as expected.

Adding a Database to our App

If you want to add data to your React app but don't want to have to build an API then this is the section for you. We'll be having a look at how we can use AWS amplify inside our react app to allow us to access our database on the back end using GraphQL.

To start we need to go into the terminal and run:

```
amplify add api
```

This will start us in a set of CLI options, asking us a few configuration questions:

What kind of API we want to use: **GraphQL**
The name of the API: **songAPI**
How we want to authenticate the API: **Amazon Cognito User Pool**
Advanced Settings: **No, I am done**
Do you have a schema: **No**
What kind of schema do you want: **Single object with fields**

```
? Please select from one of the below mentioned services: GraphQL
? Provide API name: songAPI
? Choose the default authorization type for the API Amazon Cognito User Pool
Use a Cognito user pool configured as a part of this project.
? Do you want to configure advanced settings for the GraphQL API No, I am done.
? Do you have an annotated GraphQL schema? No
? Choose a schema template: Single object with fields (e.g., “Todo” with ID, name, description)
```

After a little setup we are asked if we want to edit our new schema. We want to say yes.

This opens the GraphQL schema which we're going to update to be the schema listed here.

```
type Song @model {
    id: ID!
    title: String!
    description: String!
    filePath: String!
    likes: Int!
    owner: String!
}
```

With our schema set up we're going to run `amplify push` which will compare out current amplify setup with that on our AWS account. As we've added a new API we'll have changes so will be asked if we want to continue with the changes.

Once we've selected **Yes** then we're put into another set of options.

Do we want to generate code for our GraphQL API: **Yes**
Which Language: **JavaScript**
File pattern for the new files: **src/graphql/**/*.js**
Generate all operations: **Yes**
Maximum statement depth: **2**

This will now deploy all of the changes to AWS and also set up the new request files in our React app. This does take a few minutes to do.

Once that is completed we can go into our `App.js` file and rename it to be `App.jsx`. This just makes it easier to write our JSX code.

We now need to write a function in here to get the list of songs from our new database. This function calls the GraphQL API passing in the operation of `listSongs`. We also need to add a new state to the `App` component.

```
const [songs, setSongs] = useState([]);

const fetchSongs = async () => {
    try {
        const songData = await API.graphql(graphqlOperation(listSongs));
        const songList = songData.data.listSongs.items;
        console.log('song list', songList);
        setSongs(songList);
    } catch (error) {
        console.log('error on fetching songs', error);
    }
};
```

We now need to add or update a few imports to our file to get this working

```
import React, { useState, useEffect } from 'react';
import { listSongs } from './graphql/queries';
import Amplify, { API, graphqlOperation } from 'aws-amplify';
```

The `listSongs` is one of those functions created by amplify to help us access our data. You can see the other functions that are available in the `./graphql` folder.

Now we want this function to be called once when the component renders, but not every time that it re-renders. To do this we use `useEffect` but make sure to add a second parameter of `[]` so that it only gets triggered once.

```
useEffect(() => {
   fetchSongs();
}, []);
```

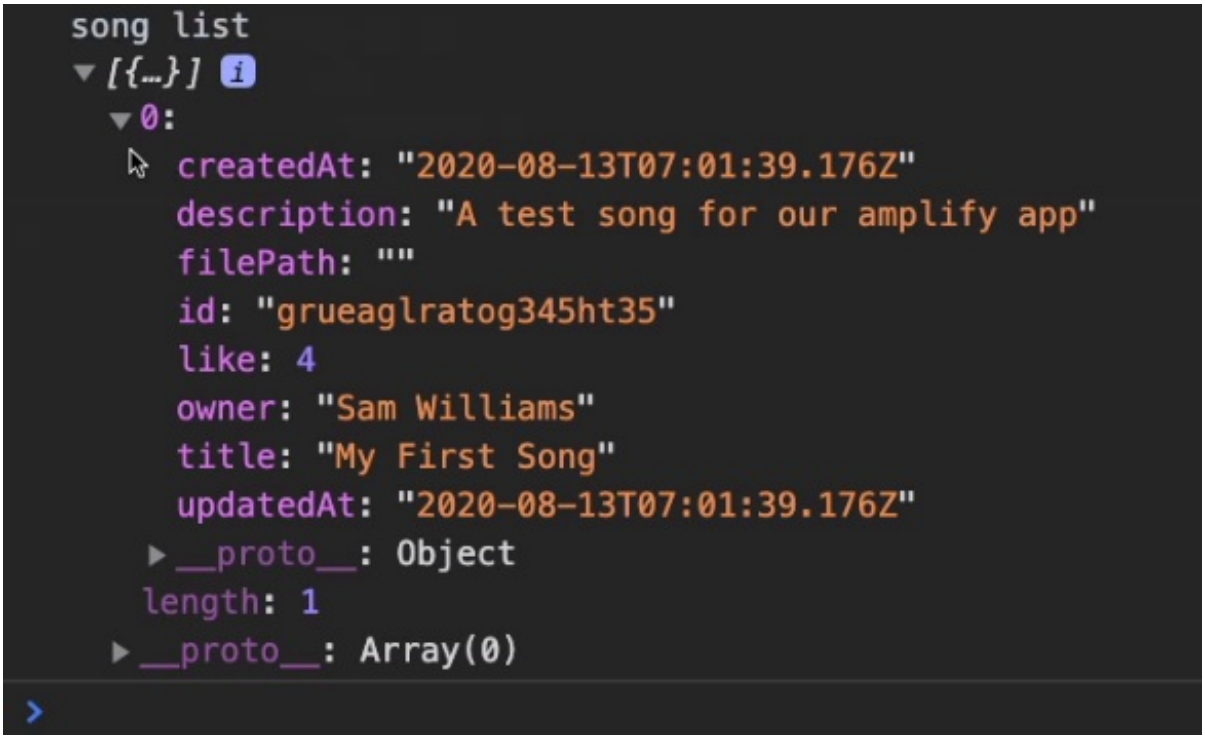
If we now start our app using `npm start` and then go to the app we can open the console and see a log of `song list []`. This means that the `useEffect` has called the `fetchSongs` which is console logging out the result, but currently there is nothing in the database.

To correct this we need to log into our AWS account and add find the DynamoDB service. We should find a new table called something like `Song-5gq8g8wh64w-dev`. If you can't find it make sure to check other regions as well. This currently has no data so we need to add some. For now we're going with manually creating new data in here. Under `Items` click `Create item` and then make sure the dropdown in the top left shows `text`. If it shows `tree` then just click it and change it to `text`. We can then make the data to go into that row.

We start with the GraphQL schema, giving the row some data for each attribute but also need to add a `createdAt` and `updatedAt` value. This can be found using the browser console and typing `new Date().toISOString()` and copying the result of that. You should end up with an object like this:

```
{
 "id": "gr4334t4tog345ht35",
 "title": "My First Song",
 "description": "A test song for our amplify app",
 "owner": "Sam Williams",
 "filePath": "",
 "likes": 4,
 "createdAt": "2020-08-13T07:01:39.176Z",
 "updatedAt": "2020-08-13T07:01:39.176Z"
}
```

If we save that new object then we can go back into our app and refresh the page. We should now be able to see our data in the console.log.



We can now use this data in our app to show the list of songs that we just got. Replace the existing text of `song list` with this set of JSX.

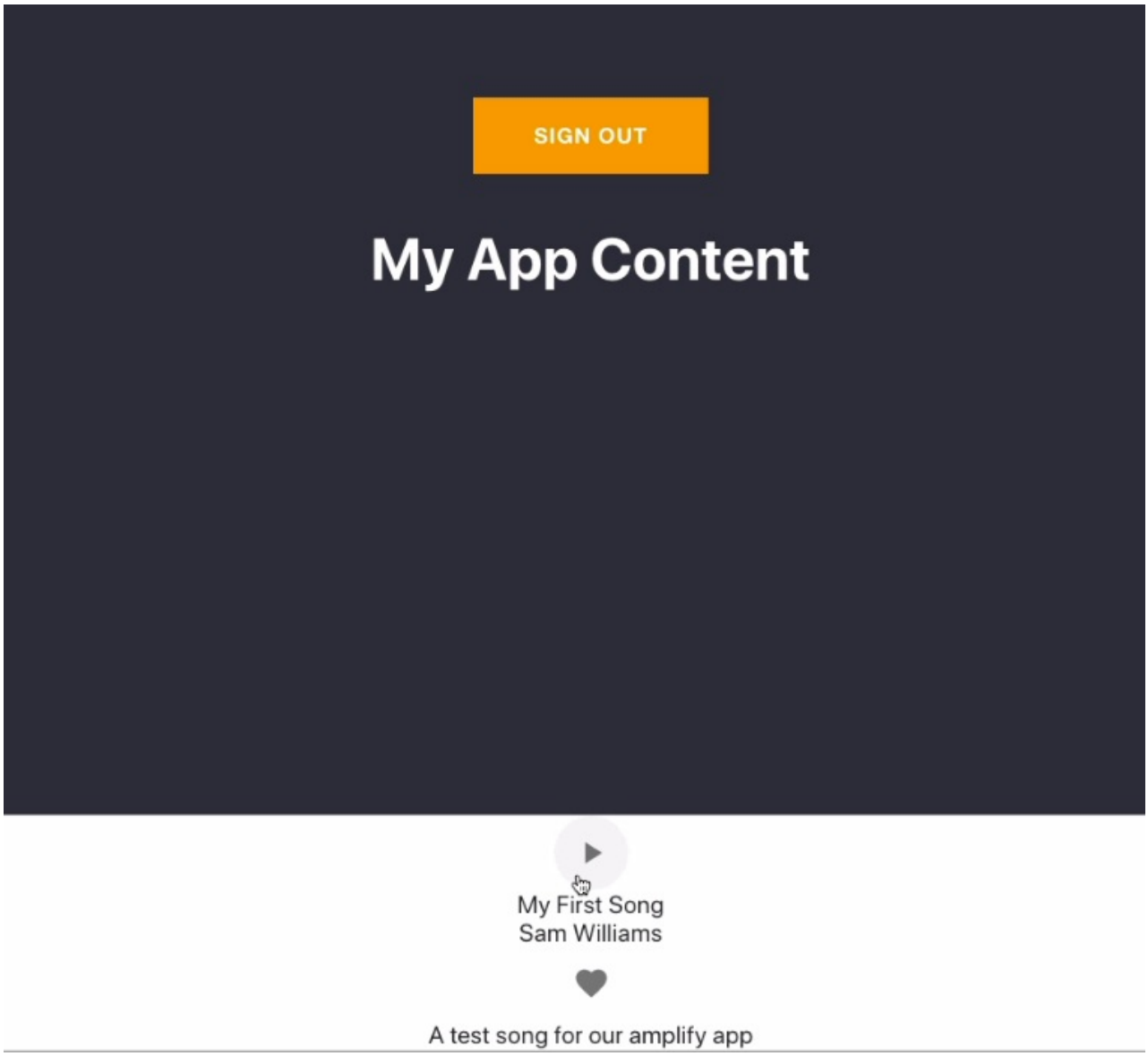
```
<div className="songList">
  {songs.map((song, idx) => {
    return (
      <Paper variant="outlined" elevation={2} key={`song${idx}`}>
        <div className="songCard">
          <IconButton aria-label="play">
            <PlayArrowIcon />
          </IconButton>
          <div>
            <div className="songTitle">{song.title}</div>
            <div className="songOwner">{song.owner}</div>
          </div>
          <div>
            <IconButton aria-label="like">
              <FavoriteIcon />
            </IconButton>
            {song.likes}
          </div>
          <div className="songDescription">{song.description}</div>
        </div>
      </Paper>
    );
  })}
</div>
```

This code is mapping over each song in the list and rendering a new `Paper` for them with all the details we need. We're using the MaterialUI library to help make this look nice for us so we need to make sure to run `npm install --save @material-ui/core`

`@material-ui/icons` to install those packages and then add them to the imports at the top of the file too:

```
import { Paper, IconButton } from '@material-ui/core';
import PlayArrowIcon from '@material-ui/icons/PlayArrow';
import FavoriteIcon from '@material-ui/icons/Favorite';
```

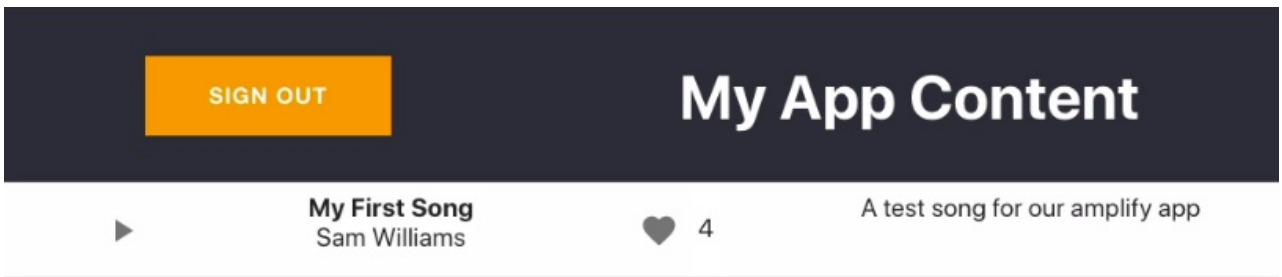
With this, if we save and reload our app we now get this:



Whilst this is ok, we can update the CSS to make it look far better. Open up your `App.css` file and change it to this.

```
.App {
  text-align: center;
}

.App-logo {
  height: 10vmin;
  pointer-events: none;
}

.App-header {
  background-color: #282c34;
  min-height: 5vh;
  display: flex;
  align-items: center;
  justify-content: space-around;
  font-size: calc(10px + 2vmin);
  color: white;
}

.App-link {
  color: #61dafb;
}

.songList {
  display: flex;
  flex-direction: column;
}

.songCard {
  display: flex;
  justify-content: space-around;
  padding: 5px;
}

.songTitle {
  font-weight: bold;
}
```

Now we get it looking like this - much better.



Now we've got one item in the database so only get one record. If we go back into Dynamo and create a new item or duplicate the existing one then we can see how multiple songs look. You can duplicate an item by clicking the checkbox to it's left

Now that we can get the data, what about updating that info. For this we are going to add the ability to like a song. To start this we can add an `onClick` function to the icon button that we have for the likes.

```
<IconButton aria-label="like" onClick={() => addLike(idx)}>
  <FavoriteIcon />
</IconButton>
```

You may have realised that there is this `idx` property that we haven't see before. That is short for index. Where we do the `songs.map` we can update it slightly to get the position of each item in the list. We can also use this idx to add a key to the top level `Paper` in that map to remove an error we get from React.

```
{songs.map((song, idx) => {
  return (
    <Paper variant="outlined" elevation={2} key={`song${idx}`}>
      ...
    </Paper>
  )}
)}
```

With the new index and the `onClick` function call we now need to make the `addLike` function. This function needs to take the index of the song to find the correct song, update the number of likes. It then removes some fields that can't be passed into the `updateSong` operation before calling that operation.

```
const addLike = async idx => {
  try {
    const song = songs[idx];
    song.likes = song.likes + 1;
    delete song.createdAt;
    delete song.updatedAt;

    const songData = await API.graphql(graphqlOperation(updateSong, { input: song }));
    const songList = [...songs];
    songList[idx] = songData.data.updateSong;
    setSongs(songList);
  } catch (error) {
    console.log('error on adding Like to song', error);
  }
};
```

Once the song has been updated in the database, we need to get that update back into our state. We need to clone the existing songs using `const songList = [...songs]` . If we just mutates the original list of songs then React wouldn't have re-rendered the page. With that new song list we call `setSongs` to update our state and we're done with the function. We just need to add one more import to the top of the file which we get from the mutators that Amplify created:

```
import { updateSong } from './graphql/mutations';
```

Now when we click on the like button on a song, it is updated in state and in the database.

Adding File Storage

Now that we have the song data stored in Dynamo, we want to store the actual MP3 data somewhere. We'll be storing the songs in S3 and accessing them using Amplify.

Watch Video At: https://youtu.be/ZpdgHjbnefo

Adding Play/Pause Functionality

To get started we're going to add a way to play and pause each of the songs. For now, this will just change the play button to a pause button. Later we'll be using the Amplify Storage methods to get our MP3 file from S3 and play it directly in our app.

We're going to add a new function to the `App` component called `toggleSong`.

```
const toggleSong = async idx => {
  if (songPlaying === idx) {
    setSongPlaying('');
    return;
  }
  setSongPlaying(idx);
  return
}
```

To get this working we also need to add a new state variable to the app. This will be used to store the index of the song that is currently playing.

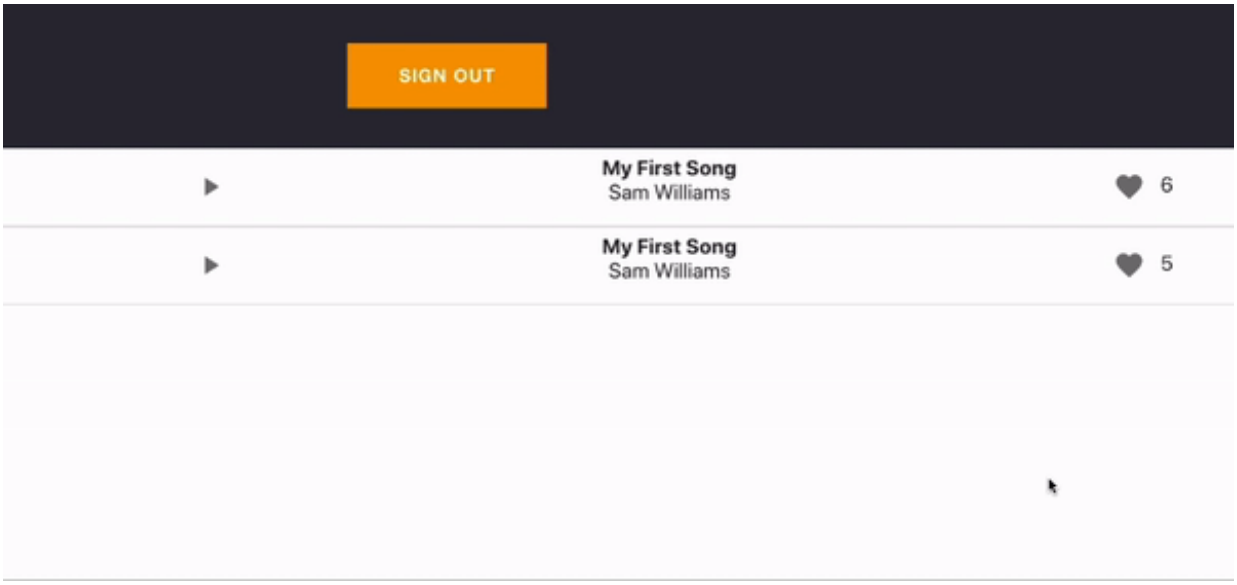
```
const [songPlaying, setSongPlaying] = useState('');
```

With this set up we need to make a change to the JSX code to use our new function and variables. Find the first button in the `songCard` div and we're going to be adding an `onClick` which calls our new function. We're also going to use a ternary equation to say that if the song that is playing is this song, show a pause icon instead of a play icon.

```
<IconButton aria-label="play" onClick={() => toggleSong(idx)}>
  {songPlaying === idx ? <PauseIcon /> : <PlayArrowIcon />}
</IconButton>
```

We'll just need to import the `PauseIcon` at the top of the file and we'll be ready.

```
import PauseIcon from '@material-ui/icons/Pause';
```



Adding the Storage to Amplify

Next, we need to use the Amplify CLI to add the storage to our app. We can start by going into our terminal and running `amplify add storage`. This will open a CLI where we need to select the following options.

Please select your service = **Content (images, audio, video, etc.)**
Friendly name for your resource = **songStorage**
Bucket name = **song-storage**
Who should have access = **Auth Users Only**
What access do those users have = **Read** and **Create/Update**
Do you want a Lambda trigger? = **No**

With that all configured we need to deploy it and we can start that with `amplify push` which will take a little bit of time to work out what we've changed in our amplify application. You'll then get a little display of all of the resources we have in Amplify. The only change is the creation of our new `songStorage` resource. We can select **Yes** to continuing and then it will create our S3 bucket for us.

Accessing the S3 File through the Amplify Storage methods

Once the deployment has finished we can start using that to access our songs. Back in our `toggleSong` function we are going to add some extra logic.

```
const toggleSong = async idx => {
  if (songPlaying === idx) {
    setSongPlaying('');
    return;
  }

  const songFilePath = songs[idx].filePath;
  try {
    const fileAccessURL = await Storage.get(songFilePath, { expires: 60 });
    console.log('access url', fileAccessURL);
    setSongPlaying(idx);
    setAudioURL(fileAccessURL);
    return;
  } catch (error) {
    console.error('error accessing the file from s3', error);
    setAudioURL('');
    setSongPlaying('');
  }
};
```

This is getting the file path from the song data (that was stored in Dynamo) and then using `Storage.get(songFilePath, { expires: 60 });` to get an access url for the file. The `{ expires: 60 }` on the end is saying that the url returned is only going to work for 60 seconds. After that you won't be able to access the file with it. This is a useful security measure so people can't share the url to people who shouldn't have access to the files.

Once we have the file, we're also setting that in a new state variable using `setAudioURL`. At the top of our `App` we need to add this extra state.

```
const [audioURL, setAudioURL] = useState('');
```

If we save this and go back into our app, if we press the play button and open the console we'll see the url being logged out. This is what we're going to use to play the song.

Uploading Our Songs

We're now getting to the point where we need some songs to play. If we go into our AWS account and search for `S3` and then search for `song`, we should find our new S3 bucket. In there we need to create a new folder called `public`. This is because the files will be public to everyone who has signed into the app. There are other ways of storing data which is private and only viewable by specific people.

Inside that folder we need to upload two songs. I have two copyright free songs that I uploaded called `epic.mp3` and `tomorrow.mp3`. Once they have been uploaded we need to update our Dynamo data to point at those songs.

In Dynamo we can find our `Songs-(lots of random charachters)` table. Under `Items` we should have two records. Open up the first one and change the `filePath` to `tomorrow.mp3` and the name to `Tomorrow`. Save that and open the second song,

changing that to `"filePath": "epic.mp3"` and `"name": "Epic"` , saving that file too. If you used your own songs then just make sure the filePath matches the file name of your songs.

We can now go back into our app, refresh the page and press play on one of the songs. If we look in our console and copy the url we're given we can paste that into a browser and our song should start playing.

Adding a Media Player to Our App

Now we want to be able to actually play our song from within our app. To do this we're going to use a library called `react-player` . We need to first install it with `npm install --save react-player` .

In our app we can then import it at the top of the file.

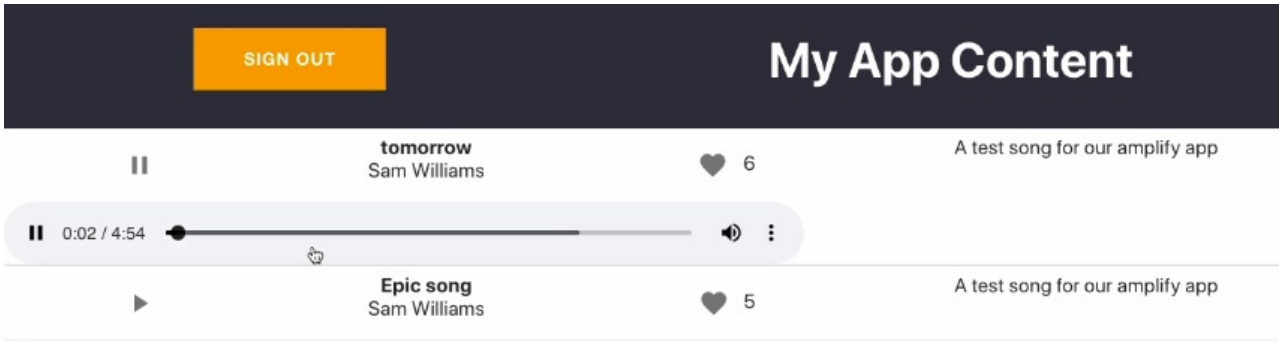
```
import ReactPlayer from 'react-player';
```

If we find our `<div className="songCard">` , we want to add our player after that component. In the same way we showed the play/pause icons, we're going to show or hide this player based on which song is playing.

```
<div className="songCard">
   .. ..
</div>
{songPlaying === idx ? (
   <div className="ourAudioPlayer">
      <ReactPlayer
         url={audioURL}
         controls
         playing
         height="50px"
         onPause={() => toggleSong(idx)}
      />
   </div>
) : null}
```

The `ReactPlayer` takes a few parameters. The `url` is just the file url to play, which is the one we get from Amplify Storage. `controls` allows the user to change the volume or pause the song, `playing` means the song starts playing as soon as it's loaded and `onPause` is a listener so we can have the inbuilt pause button work the same as our pause button.

With this all done we can save everything again and open our App. In there if we press play on one of the songs, we should see a player appear and the song will be playing.



Enabling User Uploads

Now that we have an app which allows users to view all of the songs and play them from S3. To build upon that we want to allow our users to upload their own songs to the platform.

We first need to create a way to add a song and input some information. We start by creating an `Add` button.

```
{
  showAddSong ? <AddSong /> : <IconButton> <AddIcon /> </IconButton>
}
```

We then also need to add the `AddIcon` to our imports.

```
import AddIcon from '@material-ui/icons/Add';
```

Now we can move onto creating the new `AddSong` component. We can create this at the bottom of our `App.jsx` file.

```
const AddSong = () => {
  return (
    <div className="newSong">
      <TextField label="Title" />
      <TextField label="Artist" />
      <TextField label="Description" />
    </div>
  )
}
```

We also need to add `TextField` to our imports from material UI.

```
import { Paper, IconButton, TextField } from '@material-ui/core';
```

The next thing to do is add the ability to open our new component by controlling the `showAddSong` variable. We need to create a new state declaration next to the others.

```
const [showAddSong, setShowAddNewSong] = useState(false);
```

We can now update our new `AddIcon` button to set `showAddSong` to true.

```
<IconButton onClick={() => setShowAddNewSong(true)}>
  <AddIcon />
</IconButton>
```

To change it back, we can add a parameter to our `AddSong` component called `onUpload`. When this gets called we will reset the showAddSong to false.

```
<AddSong
  onUpload={() => {
    setShowAddNewSong(false);
  }}
/>
```

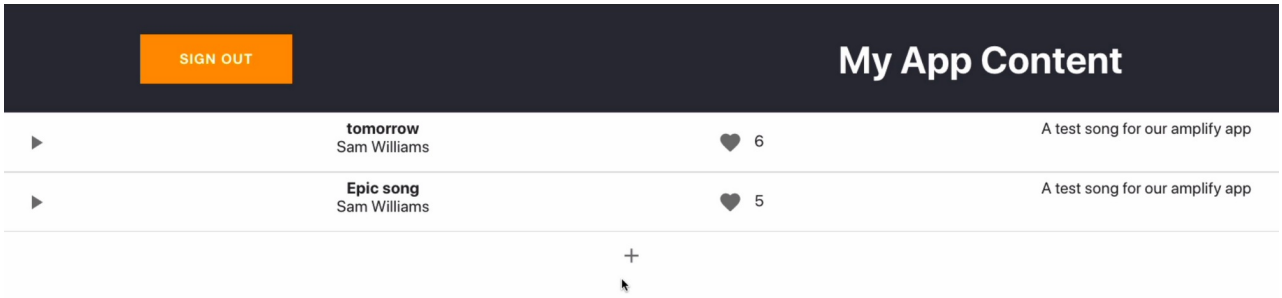
We then need to update our component to work with that new parameter and a button to "upload" the new song. That button calls a function in the component where we will add the ability to upload the data, but for now we will just call the `onUpload` function.

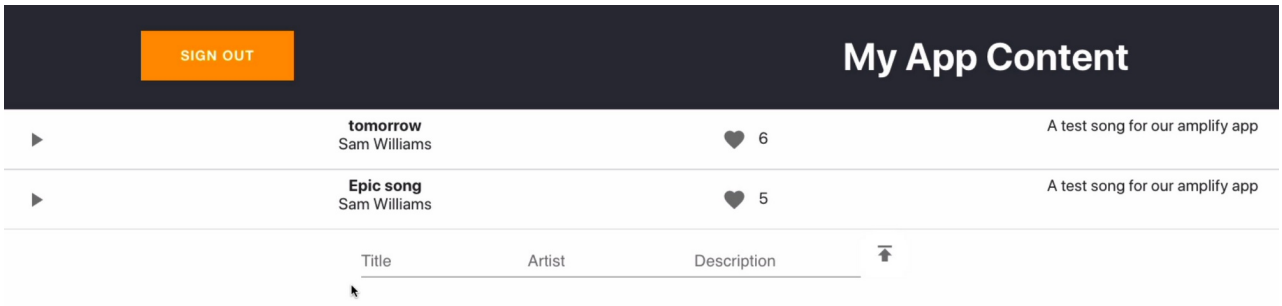
```
const AddSong = ({ onUpload }) => {
  const uploadSong = async () => {
    //Upload the song
    onUpload();
  };

  return (
    <div className="newSong">
      <TextField
        label="Title"
      />
      <TextField
        label="Artist"
      />
      <TextField
        label="Description"
      />
      <IconButton onClick={uploadSong}>
        <PublishIcon />
      </IconButton>
    </div>
  );
};
```

And now we add the `PublishIcon` to our imports and we're ready to test this out.

```
import PublishIcon from '@material-ui/icons/Publish';
```

When we start up the app and log in we now get a plus icon, clicking that we can enter some details for the song and click upload.





Updating AddSong

Now we want to be able to store and access the data that a user enters into the fields when adding a song.

```
const AddSong = ({ onUpload }) => {
  const [songData, setSongData] = useState({});

  const uploadSong = async () => {
    //Upload the song
    console.log('songData', songData);
    const { title, description, owner } = songData;

    onUpload();
  };

  return (
    <div className="newSong">
      <TextField
        label="Title"
        value={songData.title}
        onChange={e => setSongData({ ...songData, title: e.target.value })}
      />
      <TextField
        label="Artist"
        value={songData.owner}
        onChange={e => setSongData({ ...songData, owner: e.target.value })}
      />
      <TextField
        label="Description"
        value={songData.description}
        onChange={e => setSongData({ ...songData, description: e.target.value })}
      />
      <IconButton onClick={uploadSong}>
        <PublishIcon />
      </IconButton>
    </div>
  );
};
```

We've also had to change all of the TextFields to be controlled, passing in a value from out state and providing an `onChange` too. If we save this and try entering some details before uploading we should see a console.log of the details in our chrome console.

Next we need to add the ability to actually upload the song. For this we'll be using the default html input with a type of file. Add this to the JSX just before the upload icon button.

```
<input type="file" accept="audio/mp3" onChange={e => setMp3Data(e.target.files[0])} />
```

As you may have noticed we are calling `setMp3Data` on change. This is some more state in the AddSong component.

```
const [mp3Data, setMp3Data] = useState();
```

Now we have all of the data that we need, we can start by uploading the song to S3 and then the data to our database. To upload the song we're going to use the Amplify Storage class again. The fileName is going to be a UUID so we also need to run `npm install --save uuid` in our terminal and then import it at the top of our file `import { v4 as uuid } from 'uuid';`. We then pass in the mp3Data and a contentType and we get back an object with a key.

```
const { key } = await Storage.put(`${uuid()}.mp3`, mp3Data, { contentType: 'audio/mp3' });
```

Now we have the key we can create the record for the song in the database. As there may be multiple songs with the same name, we'll use an UUID as the ID again.

```
const createSongInput = {
  id: uuid(),
  title,
  description,
  owner,
  filePath: key,
  like: 0,
};
await API.graphql(graphqlOperation(createSong, { input: createSongInput }));
```

To get this to work we need to import the `createSong` mutator that was created when we created the dynamo storage with Amplify.

```
import { updateSong, createSong } from './graphql/mutations';
```

The last thing that we need to do is to make the app re-get the data from the database once we've finished uploading it. We can do this by adding a `fetchSongs` call as part of the `onUpload` function.

```
<AddSong
  onUpload={() => {
    setShowAddNewSong(false);
    fetchSongs();
  }}
/>
```

Now when we reload the page, we can click to add a new song, input the details, select our new song, upload it and then play it back from the app.

Sam Williams

I'm a problem solver who was luck enough to find coding and software development. I am self-taught and now run a Youtube Channel and consultancy company.
