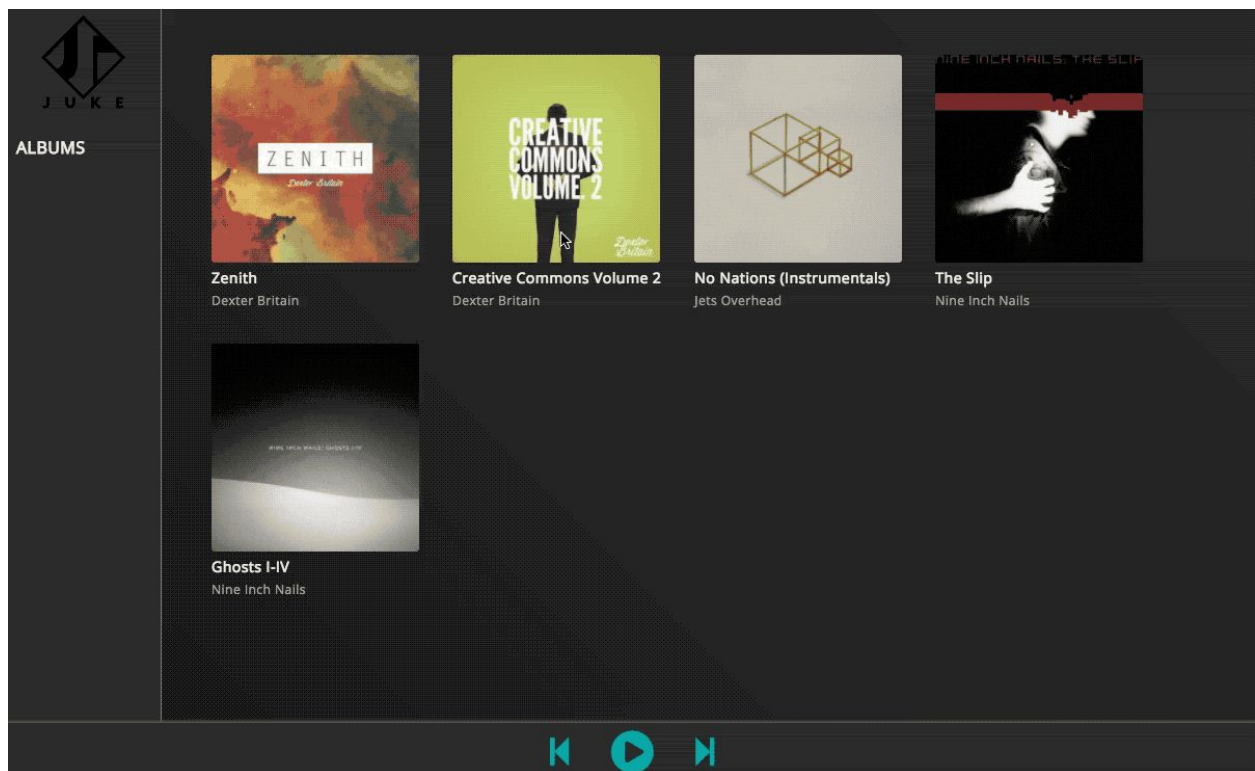


1. Intro

Goal

In this project, you will build a Spotify clone we have affectionately dubbed **Juke**. You will be given the scaffolding for a server, powered by Express and Sequelize, and the basic setup for a front-end using React. You'll also be given two HTML mock-ups and an accompanying CSS file that describe the way the app should look. As intrepid full stack engineers, your task will be to fill in the blanks to get your server running, and then craft an app where users can browse and play the songs in various albums.

WORKSHOP GOAL



Setup

- `createdb juke`
- Download zip file from Slack
- `npm install`
- For other setup instructions, refer to the `README.md`, with one exception: a seed file has been prepared for you (and can be executed by running `npm run seed` - **but don't run it just yet!** Your very first task will be to write the Sequelize models and associations that define your database tables!

2. Milestone: Routes

Goal

Our first goal is define the tables in our database using Sequelize! We will know that we have achieved our milestone when we are able to successfully seed our database using the seed file in `bin/seed.js`. You should work primarily out of the `server/db` subdirectory.

Models

Define the following models using Sequelize. We recommend making it such that each model gets its own file (ex. the Album model would go in `server/db/album.js`).

Album

- Has a `name` column with a type of `STRING`
 - `name` should be required
 - Has an `artworkUrl` column with a type of `STRING`
 - The default value should be "default-album.jpg"
-

Artist

- Has a `name` column with a type of `STRING`
 - `name` should be required
-

Song

- Has a `name` column with a type of `STRING`
 - `name` should be required

- Has an `audioUrl` column with a type of `STRING`
 - Has a `genre` column with a type of `STRING`

Associations

Now that the models are defined, we can define their associations. We'll do this in `server/db/index.js`.

- `require` each of your models into `server/db/index.js`
- Additionally, add each model to the `module.exports` of `server/db/index.js`
- Discuss the relationship between Albums, Artists and Songs with your partner. What kind of associations would make sense? After you've discussed it, consult the seed file in `bin/seed.js` and see if you can infer the associations that are expected there.

There is a one-many relationship between Albums and Songs

There is a one-many relationship between Artists and Songs

There is a one-many relationship between Artists and Albums

There are other ways that we might have chosen to organize things (for example, we could establish a many-many relationship between Songs and Artists if we

wanted to account for songs that have multiple artists, and likewise with albums), but for simplicity we'll say that each song/album has a unique artist.

A Song belongs to an Album and an Album has many Songs

A Song belongs to an Artist and an Artist has many Songs

An Album belongs to an Artist and an Artist has many Albums

To test that you have achieved the milestone, you should be able to execute the seed file in `bin/seed.js` by running `npm run seed`. If the script executes successfully and populates your `juke` database with songs, artists, and albums, then you are ready to move on to the next milestone!

3. Milestone: Routes

Goal

Now that we have a database full of data, we ought to write some API routes to serve them up! We will know that we have achieved our milestone when we are able to successfully receive our data in response to HTTP requests (which we'll make directly from the browser). You should work primarily out of the `server/api` subdirectory

Getting Albums

Define the following routes using Express. We recommend making it such that each router gets its own file (ex. the albums API routes would go in `server/api/album.js`. (For now, we'll actually only be working with the albums router).

GET /API/ALBUMS

Should respond with all of the albums in the database, including the artist for that album.

The JSON response you receive should look something like this:

```
[
  {
    "id": 2,
    "name": "Zenith",
    "artworkUrl":
      "https://learndotresources.s3.amazonaws.com/workshop/58cff0e769468300041ef
      9fd/zenith.jpeg",
    "createdAt": "2018-01-22T18:08:38.683Z",
    "updatedAt": "2018-01-22T18:08:38.683Z",
    "artistId": 1,
    "artist": {
      "id": 1,
      "name": "Dexter Britain",
      "createdAt": "2018-01-22T18:08:38.654Z",
      "updatedAt": "2018-01-22T18:08:38.654Z"
    }
  },
  {
    ...etc
  }
]</pre>
```

GET /API/ALBUMS/:ALBUMID

Should respond with a single album based on the id in the URL, including the artist and songs for that album

```
{
  "id": 2,
  "name": "Zenith",
  "artworkUrl":
"https://learndotresources.s3.amazonaws.com/workshop/58cff0e769468300041ef9fd/zenith.jpeg",
  "createdAt": "2018-01-22T18:08:38.683Z",
  "updatedAt": "2018-01-22T18:08:38.683Z",
  "artistId": 1,
  "artist": {
    "id": 1,
    "name": "Dexter Britain",
    "createdAt": "2018-01-22T18:08:38.654Z",
    "updatedAt": "2018-01-22T18:08:38.654Z"
  },
  "songs": [
    {
      "id": 13,
      "name": "Shooting Star",
      "audioUrl":
"https://storage.googleapis.com/juke-1379.appspot.com/juke-music/Dexter%20Britain/Zenith/01%20Shooting%20Star.mp3",
      "genre": "Instrumental",
      "createdAt": "2018-01-22T18:08:38.701Z",
      "updatedAt": "2018-01-22T18:08:38.701Z",
      "albumId": 2,
      "artistId": 1
    },
    {
      ...etc
    }
  ]
}</pre>
```


Using 'include' (aka eager loading)

<https://sequelize-guides.netlify.com/eager-loading/>

To test that you have achieved the milestone, make GET requests to `http://localhost:8080/api/albums` and `http://localhost:8080/api/albums/1` by entering them into your browser. If you receive the data that was specified above, then you are ready to move on to the next milestone!

4. Milestone: Routes

Goal

Now that our backend is humming along, it's time to start building our frontend!

The design team has come up with two working HTML "mocks"

(`public/juke-mock-1.html` and `public/juke-mock-2.html`) and a CSS file (`public/style.css`) that represent the way our app's two "views" should look.

To see the mock for the "all albums" view, `npm start` and navigate to `http://localhost:8080/mock-1`

To see the mock for the "single album" view, `npm start` and navigate to `http://localhost:8080/mock-2`

They look great, don't they? However, we don't want static HTML - we want a rich, interactive React app!

Our "real" single HTML page is in `client/index.html`. We've done the basic scaffolding for our app in `client/index.js` and `client/Main.js`. If you navigate to `http://localhost:8080/`, you'll see that right now we simply render a blank page.

Our goal for this next milestone (and the following milestone) is to use the static, mock HTML files to come up with nice, modular React components, and then add state and interactivity.

First, we'll focus on the components that we'll need to render all of the albums.

View for All Albums

Let's start by building out some components based on what's in the first mock file (for the "all albums" view).

- View the mock in your browser by navigating to:
`http://localhost:8080/mock-1`
- View the HTML of the mock by opening `public/juke-mock-1.html`

Work with your partner to identify a hierarchy of components in the view. Take the static HTML for each component and break them apart into separate, stateless React components. Import these components into your `client/Main.js` and use the `Main` component to put the full view together. You'll know that you've accomplished your task when navigating to `http://localhost:8080/` shows a view similar to the mock that you get from `http://localhost:8080/mock-1`.

Don't worry about adding interactivity or getting the albums from the server yet - we just want to make the view look like it does in the mock for now!

- Make sure to change all of the `"class"` properties in the HTML to be `className` in JSX!
- The Main component already starts out with a div that has the `id='main'` property, which corresponds to same wrapper div in the mock.
- Be careful to make sure that the way you wrap up the components matches the way they look in the HTML as closely as possible - otherwise, you may not get the same result from the CSS file!

It looks like it should be convenient to break the Sidebar and Player into separate components. It also looks like we could make an AllAlbums component as well. Is there anything in the HTML for AllAlbums that seems repeated? Maybe that could go in its own component too...

Identify State

At this point, you should have a view displaying the hardcoded albums from the mock. Our next step is to identify where state should "live" in our component hierarchy, and then use **dummy data** to render out that state. **Don't worry about getting the data from the server yet - render out the dummy data first!.**

In general, this is the process we want to follow when building out a React interface. Start with hardcoded JSX, then add dummy data to state and render

out the view with the dummy data, and then replace the dummy data with real data (obtained asynchronously from an AJAX request to your server). Following this process makes sure that we only deal with one problem at a time, rather than trying to tackle everything at once, which leads to mistakes.

- Identify the minimal "state" required for us to render out the view

The only thing that changes over time is our list of albums (it will go from being an empty array, to being an array full of albums). Therefore, our state should contain an "albums" array.

- Identify where this state should live

You should manage this state on your Main component. While we could start it out in our AllAlbums component, a good rule of thumb is to start your state "higher up the component tree", and then move it down as appropriate. It's generally harder to do the opposite (start with the state being managed lower down the view hierarchy, and then finding that you need to move it back up).

- Use the dummy data below to render the view with the state

```
[
  {
    "id": 1,
    "name": "No Dummy",
    "artworkUrl": "default-album.jpg",
    "artistId": 1,
    "artist": {
      "id": 1,
      "name": "The Crash Test Dummies"
    }
  },
  {
    "id": 2,
    "name": "I React to State",
    "artworkUrl": "default-album.jpg",
    "artistId": 1,
    "artist": {
      "id": 1,
      "name": "The Crash Test Dummies"
    }
  }
]
```

You will know you've succeeded when the list of albums displays using the dummy data from state!

Loading Real Data

The time has come! Let's replace the dummy albums data and render the *actual* albums saved on your server. See if you can figure it out on your own first, but the hint below will give you a more detailed approach if you get stuck.

First, we should set some initial state in our `Main` component's constructor (remember **Tom's First Law**). Because we will render once before we receive the album from the server, we need to set something there by default (much like a default function parameter) to make sure we don't get any `TypeError`s in our render.

Secondly, inside our stateful component's `componentDidMount` method we'll want to make an AJAX request for some data. Once that request completes we can call `this.setState()` to update the state with our data and trigger a re-render.

The GET request will be something like `'/api/albums'`.

Again, once the request completes, we can use `this.setState` to place the album on our state and trigger a re-render.

To test that you've achieved the milestone, refresh the page at `http://localhost:8080/` - you should see the list of albums from your server-rendered out, with the appropriate cover art

5. Milestone: Routes

Goal

We've got all of the albums displaying, so let's turn our attention to the "single album view"

- To see the mock for the "single album" view, `npm start` and navigate to `http://localhost:8080/mock-2`

When someone clicks on an album, they should see that album and all of its songs. Conceptually, it's like "all albums" and "single album" are different "pages" - but remember that we're building a single page app! This means that rather than navigating to a different page, we're just using JavaScript to swap different components in and out, based on the current state.

Our process for implementing this will be similar to what we did to implement the "all albums" view.

View for Single Album

Let's revisit our mock:

View the mock in your browser by navigating to:

`http://localhost:8080/mock-2`

View the HTML of the mock by opening `public/juke-mock-2.html`

Work with your partner to identify a hierarchy of components in the view, and then write stateless React components for them. Note that many of the components you'll have identified when composing the "all-albums" view are still there (like the sidebar and the player) - you just need to spot the components that are unique to the "single album" view.

To test it out, import the component for your single album view into `client/Main.js` and swap it in place of your "all albums" component (but be sure to put it back when you're done).

Once again, don't worry about adding any behavior yet - we just want to make the view look like it does in the mock for now!

It looks like there should be a `SingleAlbum` component. The JSX for showing the album's name and artwork looks familiar...

It seems like it would also be convenient to have a `Songs` component, to keep our `SingleAlbum` component from getting too big.

Selected Album State

Let's continue to follow our process to implement a view switch to the "single album" view. Our first goal is to make it so that the single album component displays a dummy album's data whenever we click *any* album.

Identify how we should represent state and where it should live

Your first inclination may be to have some bit of state representing what "page" we're on (i.e. `currentPage: 'single-album'`, Or `currentPage: 'all-albums'`).

This could work, but we also need to represent which album is selected - as long as we're doing that, let's use that bit of state to calculate which view we should see.

For example, we could define a `selectedAlbum` state, which contains the data for the selected album. If no album is selected, `this.state.selectedAlbum` would be an empty object. If an album is selected, then `this.state.selectedAlbum` will be populated with album data. If this is the case, we can just check the truthiness of a field that a selected album will always have (like its id) to determine whether to show all the albums or just the single album.

To go with this approach, initialize a new property like `selectedAlbum` on the state object in your Main component. Don't forget Tom's First Law! Initialize it to be an empty object (since we'll be storing an object there later)!

- Write a method on your stateful component that will update your state to reflect that a certain album has been selected. Again, use the dummy data below for now - don't worry about going to your server yet!

Here's a single dummy album: note that a single selected album has more data (the list of songs) than the albums you've received from your server!

```
{
  "id": 3,
  "name": "Chain React-ion",
  "artworkUrl": "default-album.jpg",
  "artistId": 1,
  "artist": {
    "id": 1,
    "name": "The Crash Test Dummies",
  },
  "songs": [
    {
      "id": 13,
      "name": "Set Some State",
      "audioUrl":
        "https://storage.googleapis.com/juke-1379.appspot.com/juke-music/Dexter%20
        Britain/Zenith/01%20Shooting%20Star.mp3",
      "genre": "Instrumental",
      "albumId": 2,
      "artistId": 1
    }
  ]
}
```

- Pass this method down as a prop to the component(s) representing your list of albums, and attach it as an `onClick` listener to each album (we recommend attaching it to the `<a>` element)
- Update the render method in your Main component so that you switch the view based on the presence or absence of a selected album

We want to be able to conditionally switch between the `AllAlbums` component or the `SingleAlbum` component. We can calculate this in our render based on whether or not our "selectedAlbum" is empty or not (checking for the truthiness of a field like `id` should suffice).

Normally, this would be a job for an if statement, but unfortunately we can't use an if statement directly in a JSX expression - however, we can use a ternary control operator! Do you remember how those work? [Check out these docs](#) for a reminder!

If everything is hooked up properly, clicking on any album should swap the view and show the single album component, which will render out the dummy data.

Fetch Single Album

As you may have noticed, it doesn't look like we have all of the information we need to display the songs in the albums array that we already have on state (check out `http://localhost:8080/api/albums`).

However, check out what happens when we try to fetch just one album (`http://localhost:8080/api/albums/1`, for example). That looks much better!

Modify your click handler to fetch the album from the server and put that on the state instead of the dummy data!

If you have trouble check the next page...

```
// make it so that the method we invoke when we choose an album accepts
the album Id,
// and use that to fetch the appropriate album from the server
handleClick (albumId) {
  axios.get('/api/albums/' + albumId)
    .then(** you do the rest */)
}
```

The solution is below...

```
handleClick (albumId) {  
  axios.get(/api/albums/${albumId})  
    .then(res => res.data)  
    .then(album => // Using arrow functions is important here! Otherwise,  
our this context would be undefined!  
      this.setState({ selectedAlbum: album })  
    );  
}
```

De-Selecting an Album

Awesome - our view is switching! But...we can't seem to go back to the all-albums view without refreshing. Let's make it so that if we click the "ALBUMS" link in the sidebar, we'll show all of the albums again. Try to figure it out on your own - the hint below will guide you if you get stuck, though.

- Write a new method on your `Main` component that will "reset" the `selectedAlbum` to its initial state.
- Pass that method down as a prop to your `Sidebar` component (and don't forget to `bind`).
- Attach the prop as a click handler to the appropriate JSX in the `Sidebar` component

Once you've done this, you'll have achieved the milestone!

To test that you've achieved the milestone, click on an album - you should be shown the single album component, which displays the album's artwork and a list of all of the album's songs. Clicking the "ALBUMS" link in the sidebar should return you to viewing all of the albums.

6. Milestone: Routes

Goal

If you've gotten this far and understand the concepts we've covered up until now, then great job! I'd say you're in pretty good shape with the fundamentals of React!

The final milestone in this project is to actually get your app to play some music! It will be a bit harder and there will be less guidance - be sure to write a help ticket if you get stuck! Good luck!

Play Audio

Let's pound out some tunes! Be a dear and use headphones to keep from disturbing your neighbors.

HTML5 comes with an `<audio>` element, which we could use like so:

```
const audio = document.createElement('audio');
audio.src =
'https://learndotresources.s3.amazonaws.com/workshop/5616dbe5a561920300b10cd7/Dexter_Britain_-_03_-_The_Stars_Are_Out_Interlude.mp3';
audio.load();
audio.play();
// for now, reload the page if you want to stop the music
```

In fact, start by making that one specific song play

('https://learndotresources.s3.amazonaws.com/workshop/5616dbe5a561920300b10cd7/Dexter_Britain_-_03_-_The_Stars_Are_Out_Interlude.mp3') whenever *any* of those s get clicked. Don't worry about playing the correct songs just yet, just get those buttons all doing this one thing.

HINT 1:

Try attaching an `onClick` listener to each button.

HINT 2:

`onClick` causes a function to be invoked when that DOM element is clicked. This can include accessing (or calling) a method on a component. If we wanted a component to invoke a method on its parent, we could just pass the method down from the parent to the child via props (making sure to bind the `this` context to the parent component!)

HINT 3:

In your main component you'll want something like:

```

// declare our audio as global to the module, so we can use it
const audio = document.createElement('audio');

class Main extends React.Component {
  /* ... */
  constructor (props) {
    super(props);
    // We're passing this.start to a click listener, so we need to make
    sure to bind it
    // to preserve this "this" context within the function.
    // We're not currently using "this" in this.start, but we might want to
    later...
    this.start = this.start.bind(this);
  }

  start () {
    audio.src =
'https://learndotresources.s3.amazonaws.com/workshop/5616dbe5a561920300b10
cd7/Dexter_Britain_-_03_-_The_Stars_Are_Out_Interlude.mp3';
    audio.load();
    audio.play();
  }

  render () {
    /** some jsx /
    <SingleAlbum start={this.start} />
    /** more jsx */
  }
}

```

AND THEN ADD SOMETHING LIKE THIS FOR EACH BUTTON

```
onClick={this.props.start}
```

The Right Song

Now make it so that each play button will play the song it *actually* corresponds to. Just focus on playing for now, we'll get to pausing soon enough.

...

Easier said than done, huh? There are two big steps:

1. Locate each song's audio source.

Each song as an `audioUrl`. If the `audio.src` is set to that, it will request the audio appropriately when you `audio.load` it.

2. Make it so that clicking "play" will cause the correct song to play.

We could pass the handler function directly to the `onClick` listener like so

```
<button onClick={this.props.start}>
```

The problem here is we can't pass any arguments to our "start" method. However, we could also go right ahead and pass an anonymous function to `onClick` as well

```
<button onClick={() => this.props.start()}>
```

Now, we can pass in whatever arguments we want!

The Current State of Things

Sorry, still not doing pausing. Instead, let's work on giving our users some visual feedback when they play a song. Step one: we need a way to tell that a song is even currently playing!

Remember - the *only* way to cause the render method to execute again is by using `setState`. This means that we need to find a way to represent this on our state object.

1. Take a moment to discuss with your partner how you could represent a song being played in your state.


Suggestion: We could represent this with a field called `currentSong`, that stores the selected song object, or the selected song id.

2. Add the appropriate field to your `state` object.

Remember earlier how we did `this.start = this.start.bind(this)`? If you use `this.setState` inside of the `start` function, it's important that `start` keeps the original `this` even though it's being passed around to other components. This is a pretty common problem in React and making sure that you're managing `this` properly is important.

Visual Feedback

We're aiming for something like this:

	#	Name	Artist	Genre
	1	I Should Be Born (Instrumental)	Jets Overhead	Rock
	2	Heading For Nowhere (Instrumental)	Jets Overhead	Rock
	3	Weathervanes (Instrumental)	Jets Overhead	Rock
	4	No Nations (Instrumental)	Jets Overhead	Rock

1. Let's start by hiding the play button for that song whenever a song is playing.

Remember that you can interpolate JavaScript expressions into JSX, like [ternary control operators](#).

2. Now let's make the background of a song's row highlight when the song starts. Our designers have made this convenient for us by including a `.active` class in our `style.css` file. Open up the element inspector and CTRL + click on a then "Edit as HTML". Add the `active` class and then watch the CSS magic.

Now, how to work this into our render logic? Did you know that you can give `className` an interpolated expression that evaluates to string?

```
songs.map((song) => {  
  return (  
    <tr className={song.id === this.props.currentSong.id ? 'active' : ''}>  
    //etc...  
  )  
})
```

3. While you're at it — if you haven't already — make it so that only one song can be playing at any given time. Don't worry about a song ending or anything like that, just make it so that if the user clicks a song, it stops the previous one and plays the new one.

Player

Let's open up our `Footer` component and get this party started! Here's a checklist of things to do to help you along:

- Only show the player controls once a song starts playing
- Switch the play button to a pause button once a song starts playing
- Make it so the pause button actually pauses
- Toggle the pause button back to play after pause is clicked
- Check off the checkmark to the left of this text

Try not to look at these hints (and they are only hints, not solutions) unless you get stuck:

Hint: Player controls only visible when playing

Don't you have some kind of state property that can come in handy? We could probably pass it down to the component that needs it...

Hint: Switch play to pause while song plays & vice-versa

It sounds like there's a couple factors here. Is any song playing, is this specific song playing, what causes a song to become playing or not, etc. You will need some state to hang around and check against. Note that the icons render as a result of the presence of class names provided by the [font-awesome](#) library.

Hint: Pausing

You'll need to use the `audio` HTML5 element API. Multiple functions will have to have access to this object so they can call its various methods when certain actions occur.

Next & Previous

Now make the `and` buttons work.

Start by focusing on the common cases (middle songs), then go ahead and add in wrap-around (for edge cases). Clicking `on` the last song should start the first, and clicking `on` the first should start the last.

You can also make it so that when a song ends, the next one begins to play.

HTML `<audio>` elements have an `'ended'` event we can hook into via `.addEventListener`. Go ahead and try it out - do you know where to add an event listener?

SOLUTION ON THE NEXT PAGE

```
// in your Main component
...
componentDidMount () {
  /** ... /
  audio.addEventListener('ended', () => {
    this.next(); // or some other way to go to the next song
  });
  /* ... */
}
...
```

Exit Music

ReactJS is a powerful frontend library that allows you to calculate your UI in a functional style. Think back to when we were calculating our UI in TripPlanner using the DOM API - it quickly became difficult to manage the state of the application (even though it was quite small). By separating stateful components from presentational components and reducing the number of ways to trigger a mutation to one (`setState`), reasoning about our view becomes as simple as reading a series of functions.

That being said, our code right now isn't perfect - our main component is getting fairly large, and as we introduce more and more features, it could get bloated. It's also a pain to have to pass down a prop from our main state container all the way to the bottom of our view tree - in a larger app, this could be quite far.

Fortunately, React is a lightweight, un-opinionated library - we'll soon see that there are many solutions within the React ecosystem that can help us with these problems.

Progress Rubric

This workshop is a plunge into many facets of React, forming the foundation for all to come. If you don't feel you grasp all the complexities yet, start panicking. Wait no, the other other thing, don't sweat it - totally natural and expected. Below, we'll break down what we think are the important takeaways from this workshop.

Primarily, state and props are fundamental building blocks of any React app, so it is important to gain a working understanding of them ASAP. Please connect with a fellow or instructor if this part of the puzzle seems opaque to you.

Main Takeaways

- React fundamentals
 - Front-end view library
 - Motivation & goals
 - The principles of unidirectional data flow
- React components

- What they are
- How to define them
- How they relate to the DOM
- Component lifecycle
- How and why to implement AJAX, event listeners and other side effects in `componentDidMount`
- Stateful components
 - What distinguishes a stateful component from a stateless component
 - How to initialize state in a constructor
 - The usage of `this.setState`
 - Binding the `this` context of a method
 - How a stateful component communicates with stateless components (props)
- Stateless components and JSX
 - How to receive props
 - How to implement view logic using interpolated expressions
 - How to refer to HTML attributes in JSX
 - How to attach listeners (such as `onClick`) to a JSX element
 - How to manage classes and inline-styles with JSX
 - How to transform an array of data into JSX using `.map`, and why to use the `key` prop

Resources

<http://busypeoples.github.io/post/react-component-lifecycle/>